(2)

PROCEEDINGS OF THE FIFTH ANNUAL ADA SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) SYMPOSIUM

Sponsored by:

Ada Software Engineering Education and Training Team

Ada Joint Program Office

August 14-16, 1990

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1990	Proceedings

4. TITLE AND SUBTITLE

Proceedings of the Fifth Annual Ada Software Engineering Education and Training (ASEET) Symposium

5. FUNDING NUMBERS

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Software Engineering Education and Training Team

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
The Pentagon, Rm. 3E114
Washington, D.C. 20301-3080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document contains papers and panel position papers that would have presented or distributed at the August 1990 Symposium which was canceled. Panel discussions on 1) Curriculum Trends in Software Engineering; 2) Lessons Learned in Software Engineering taught with Ada; 3) The Future of Ada. Papers presented suchaa, Implementation of Artificial Systems in Ada; A Design Methodology for Object Based Languages; Program Development and Ada; STANFINS Redesign Subsystem II: Developing a Large MIS Application Using Ada; The Place of Ada in an Undergraduate Software Engineering Curriculum; Using Short Laboratory Exercises to Develop Ada Awareness.

14. SUBJECT TERMS

Object Based Language, Artificial Intelligence, STANFINS, MIS applications, Program development, Realtime programming

15. NUMBER OF PAGES

125

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None

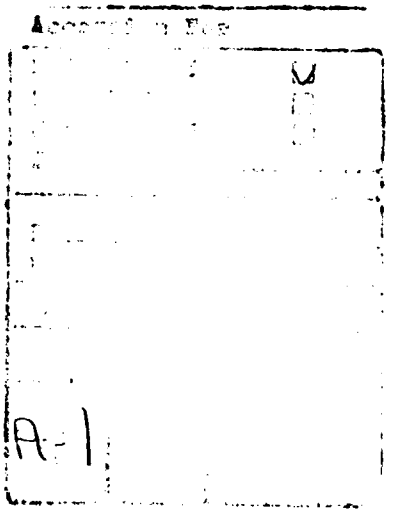
PROCEEDINGS OF THE FIFTH ANNUAL ADA SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) SYMPOSIUM

Sponsored by:

Ada Software Engineering Education and Training Team

Ada Joint Program Office

August 14-16, 1990



The views and opinions herein are those of the authors. Unless specifically stated to the contrary, they do not represent official positions of the authors' employers, the Ada Software Engineering Education and Training Team, the Ada Joint Program Office, or the Department of Defense.

ASEET TEAM MEMBERSHIP LIST

Ms. Wanda B. Barber
USA Information Systems Software Development Center-Lee
Stop L-75
Fort Lee, VA 23801
wbarber@ajpo.sei.cmu.edu

Captain Roger Beauman
HQ ACD/AULP
ATT: Captain Beauman
Scott AFB, IL 62225
beauman@ajpo.sei.cmu.edu

1LT Sandra Chandler
Software Engineering Training Branch
3390 TCHTG/TTMKPP
Keesler AFB, Mississippi 39534-5000

Captain David A. Cook
1403 Francis Drive
College Station, TX 77840
dcook@ajpo.sei.cmu.edu

Mr. Leslie W. Dupaix
USAF Software Technology Support Center
OO-ALC/MMEA-1
Hill AFB, Utah 84056

Major Charles B. Engle, Jr. (Ret)
Florida Institute of Technology
Department of Computer Science
150 West University Blvd.
engle@cs.fit.edu
engle@zach.fit.edu
Melbourne, FL 32901-6988

1LT Dan Herod
HQ ATC/SCDBT
Randolph AFB, Texas 78150
scdbt6latc136@ran314.af.mil

Captain Joyce Jenkins
US Air Force Academy
Computer Science Department
Colorado Springs, CO 80840
jenkins@usafa.af.mil

Major Jay Joiner
Department of Computer Science
USAF Academy, CO 80840
joiner@usafa.af.mil

Major Pat Lawlis
AFIT/ENG
Wright Patterson AFB, Ohio
lawlisp@ajpo.sei.cmu.edu
lawlis@blackbird.afit.af.mil

Ms. Cathy McDonald
IDA
1801 N. Beauregard Street
Alexandria, VA 22311
mcdonald@ajpo.sei.cmu.edu

LCdr Lindy Moran
PACOPSUPPFAC
Box 9
Pearl Harbor, HI 96860-7150
moranl@ajpo.sei.cmu.edu

Lt Darin Morrow
Software Engineering Training Branch
3390 TCHTG/TTMKPP
Keesler AFB, Mississippi 39534-5000

E.K. Park
Assist. Professor301-267-3080
Computer Science (M.S. 9F)
U.S. Naval Academy
Annapolis, MD 21402
eun@USNA.MIL

Major Doug Samuels
Headquarters AFSC/PLR
Andrews AFB, Maryland 20334-5000
dsamuels@ajpo.sei.cmu.edu

Captain Michael Simpson
OTS/RMC
Lackland AFB, Texas 78236-5000
msimpson@ajpo.sei.cmu.edu

Major David Umphress
Air Force Institute of Technology
AFIT/ENG
WPAFB, OH 45433
dumphres@blackbird.afit.af.mil
dumphres@galaxy.afit.af.mil

This Page Left Blank Intentionally

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

MESSAGE FROM THE SYMPOSIUM CHAIR

Catherine W. McDonald

The following pages include the papers and panel position papers that would have been distributed in August 1990 as part of the proceedings for the Fifth Annual ASEET Symposium. Because of financial constraints, this symposium had to be cancelled. However, the ASEET Team was able to print these proceedings and make them available to the Ada education and training community. A special thanks goes out to all the authors who graciously agreed to the publication of their papers even though they were not able to present their papers at the symposium.

I would like to take this opportunity to thank all the people who have supported the Team over the years and offered special support during this past year. The Team, under the auspices of the AJPO, will continue to work to meet the needs of the DoD Ada education and training community and, hopefully, to conduct future workshops and symposia.

MESSAGE FROM THE ADA SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) TEAM CHAIR

Major Doug Samuels, USAF

Welcome to the Fifth Annual ASEET Symposium. This year we have endeavored to provide a wide range of speakers and papers from industry, academia, and government. Interaction and the exchange of ideas are paramount to the successful instantiation of Ada in these domains. This symposium provides an excellent opportunity to accomplish this goal. Our exhibitors bring you education and trainingmaterials/software tools from the leading edge of their fields. Please take the time to find out what they have to offer. There are critique forms throughout the symposium. Please fill these out for they provide the only medium through which we can improve our symposia.

This symposium is the last on at which I will be the ASEET Team Chair. It is time for new blood to surge through the team's veins. I have enjoyed sharing our symposia and workshops over the last three years. I am certain the next chair, Maj David Umphress, will enjoy future ones. I wish to thank the members of the team, whose untiring efforts have made the team the huge success it is, and, of course, I thank you, supporters of the team's efforts and endeavors.

A Design Methodology for Object-based Languages

Jaime Nino
Department of Computer Science
University Of New Orleans
e-mail : jncs@uno.bitnet

Abstract

The development of modular languages, such as Ada, was intended to facilitate the production of software systems which support recognized software engineering principles amenable to the production of systems of a high degree of understandability and reusability. To ensure that these benefits are realized in such systems, however, requires more than just a programming language. By enlarging the abstraction design unit, we propose an object-based design methodology which in particular is aimed to be used by beginner Ada programmers. This methodology is a composition of the top-down design methodology and the data-driven approach.

Introduction

The development of the Ada language was intended to facilitate the production of software systems that are more reliable, modifiable, efficient and understandable. To ensure that these benefits are realized in Ada systems, however, requires more than just a programming language. A coherent software engineering methodology that guides the structuring of systems in a disciplined manner, a methodology that is replicable and that takes full advantage of Ada features is needed.

Previous experiments have demonstrated that improved software quality is not an automatic by-product of the use of Ada. In a 1982 case study, a traditional, top-down functional design methodology was used in redesigning a software system in Ada. The software had initially been implemented in FORTRAN. Although the Ada programmers were unaware of the FORTRAN design the finished Ada system was remarkably similar to the original FORTRAN system. It was concluded that few of the potential benefits of Ada had been realized. What is needed is a design methodology that exploits Ada's rich abstraction mechanisms, among them those supporting information hiding, encapsulation and software reusability.

By far the most successful design methodology used for procedural languages is top-down design. This methodology is based on successive refinement of functional specifications. Among the recognized positive characteristics of this methodology are :

i. *Design correctness* : Functional correctness of resulting design.
ii. *Design composition* : Lower level refinements are completely specified by higher level ones.
iii. *Design implementation* : Using stubs, we can alternate our activities between design, implementation and testing in that order.

Ada separate compilation and subunit features do directly support the top-down design

methodology. Unfortunately, top-down design undermines the use of Ada features available to enhance maintenance, extendibility and reusability of software. To wit,

i. *Encapsulation* : functional decomposition uses procedures and functions as its principal tool for refinement and implementation. The data manipulated and their respective implementation play only a secondary role, being the support of the implementation of the design. Consequently, only limited data encapsulation can be achieved.

ii. *Genericity* : Lower level refinements are specified and tailored by and for the support of higher level refinements. This can lead to overspecification of design components.

It is clear that this methodology can not be used for Ada without modification if we wish a full utilization of its features. These were designed to support well known software engineering principles mandated by current software requirement and software evolution. The software engineering concepts needed to support top-down design such as procedural abstraction, structural design, and information hiding are at a lower level of abstraction necessary to take full advantage of higher abstraction features as presented in Ada. Furthermore these procedure-oriented methodologies work well when applied to the small, well-defined, well constraint problems, but are simply no longer adequate for dealing with large-scale, dynamic commercial and industrial software systems

Object-base languages

For the purpose of discovery of an appropriate software design methodology for the Ada language, we must identify a different abstraction unit for design, composition and evolution of software systems; a unit at a higher abstraction level from the functional/procedural unit. The abstraction unit which we will use for the methodology proposed in this paper, called *object*, is the fundamental unit for system composition used and supported by the so-called Object Oriented Programming Languages. Objects, each with its own local state and operations to manipulate this state, are used to model data type values and their functionality.

In [19] Wegner classified object-based languages as being those that support objects as a language feature. In this paper we adhere to this classification. Object-based characterization is seen as a necessary but not sufficient requirement for being object-oriented. This latter category requires modules to have a inter-module structure besides from being library units. See [8,12,14,15]

Within object-based languages, a collection of interacting objects constitute a system. For system decomposition, objects are to object-based designers as subprograms are to Pascal programmers. Design based on objects, called *object-based* design, consists of decomposition of the system into self-contained objects which interact via a well defined interface. A software designer, using object based design, has as ultimate goal the production of a simulation of the desired software system based on objects. Thus we cannot distinguish designing and coding as being completely separate activities. Using object-based languages, designers are programmers at a very high level because they specify the objects that will constitute the system. This specification is done by defining the internal semantics of the object as well as its external interface.

There are a number of software metrics used for design and implementation evaluation of a system. These metrics include being able to reuse, refine, test, maintain and extend existing code. It is noted that the value of these metrics have been decreasing as the size of applications, and hence their complexity, has been increasing.

A major benefit of object-based programming is found in its ability to increase the value of these metrics. Object-based programming does this by increasing the ability to manage complexity of software development. The most effective tool for dealing with complexity is abstraction; object-based design supports abstraction via separation and encapsulation of concerns, which in turn allow us to manage complexity.

Programming in an object-based language, however, does not ensure that the complexity of an application will be encapsulated. Applying sound programming techniques can improve encapsulation, but the full benefit of object-based programming can be realized only if encapsulation is a recognized goal of the design process.

The main goal of this paper is to present an object-based design methodology which in particular is aimed to be used successfully by beginner Ada programmers.

Structure and Semantics of Objects

The driving motivation behind object-based modelling is to match the technical specification and representation of a system more closely to the conceptual view of the real world. An object is an abstraction and an encapsulation; an abstraction of the problem space representing a well defined entity within it; as an abstraction the object model is focussed on the external view of an object, as it is perceived functionally by its users. In this fashion objects are tools for separation of concerns, in this case essential behavior from its numerous possible implementation choices. An object is an encapsulation of its state, attributes and functionality; methodical encapsulation localizes design decisions that can be characterized as being temporal, overspecific, or secondary to the nature of the object.

From the point of view of abstraction an object is :

- tangible/visible entity from the problem space
- Entity that plays a well-defined role in the problem space
- Entity that can render a service.
- model of reality

From the point of view of object modeling within a programming language, the fundamental components of an object are :

- State
- Functionality
- Identity

The state of an object includes all the static attributes that characterize it, as well as the current values of these attributes. Attributes can be objects themselves. These attributes are

updated and managed by the object itself. Other objects are allowed to look-up and update this state via a well defined functional interface provided by the object.

The functionality of an object includes all the operations needed to manipulate its state and perform operations (services) for other objects. These operations have specific semantics based on the object behavior being model. These operations include :

- Constructors/Destructors
- State Modifiers
- Selectors
- Iterators
- Input, output
- Service-rendering

The Identity is the uniqueness condition of an object which separates and identifies it from all other objects. (Object identity plays a major role in Object-Oriented Databases).

We can readily infer that the object-based model is fundamentally different than the models embraced by structured analysis, structured design and structured programming methodologies. Furthermore it has been widely recognized that the benefits of the object model address fundamental needs of large software system development and system revision and enhancement. These include:

- Well-structured complex systems by using modules as units of design as well as encapsulation and information hiding.
- Full use of the expressive power of object-based languages.
- Encourages software reuse
- Leads to systems that are malleable to system evolution
- Reduces the risk factor in the development of complex systems
- Reduces development time
- Results in smaller programs

The object-based model is very useful in the modeling of the *server/client* paradigm. In this paradigm one or more clients request services to be performed by a server. This one in turn knows the services it can perform and can test its own state for the feasibility of a service.

Design methodologies for object-based languages

Several object-based design methodologies have been proposed [1,2,3,4,5,6,12,19] which partially address issues of object design. These methodologies emphasize the proper design of objects to ensure completeness in the object specification and encapsulation, as well as to ensure reusability; in fact they fail to offer complete methodologies as they concentrate only on few aspects of the design spectrum. Moreover, they do not take into consideration the most prevalent form of software specification. Software specification is given via a functional description about the desired behavior of the software system. Thus a designer for an object-based language starts with descriptions about behavior and functional transformations of input data to output data and must produce objects which model the system in such a way that their join functionality produces

the specified behavior.

In [3] Booch introduced a methodology tailored for Ada, by recognizing the need of a methodology around objects. The steps of Booch's object oriented technique are simply :

- define the problem
- develop an informal strategy
- formalize the strategy
- identify object and their attributes
- identify operations on the objects
- Establish interfaces
- implement the operations

This methodology is complemented with a design taxonomy introduced by Booch [4] for reusable software based on generic packages, and which reflects the concerns and needs of many software developers. It is a taxonomy which is based on actual Ada development experience and is already in use in a number of different organizations. It deals not only with the functional behavior of a software component, but also with its temporal and spatial behavior.

Another methodology, responsibility-driven methodology presented [19], has as its goal the improvement of data encapsulation. Like other proposed methodologies it is based on object design by stressing the fact that objects are tools for modeling the server/client paradigm.

A major common short-coming found in these methodologies is the lack of a well defined approach for object identification. Although these methodologies propose guidelines for this task, they can be effectively used only in the hands of an experienced designer; and as guidelines they fall short from being a methodology that can be successfully used and can be replicated by all object-based designers including beginners. They also fail in providing a methodology to guarantee and determine functional correctness and completeness of the resulting design. Software design demands a rigorous, replicable methodology to ensure success to its users.

A Pragmatic View into Object-based Design

Before we discuss our proposed methodology, we describe the ad-hoc methods used successfully by many programmers experienced with object-oriented programming languages. These programmers have a consistent set of techniques for building systems centered around a few key ideas. First is the importance of *iteration, revision, and refinement.* Very early in the life of a project, design centers around the identification of the correct objects to simulate the application at hand. Objects are created, operations written, and code produced primarily for purposes of design and prototyping, to identify the correct set of objects and the proper distribution of functionality among them. The designer/programmer is expected to completely revise the design several times until a satisfactory set of objects is discovered.

Another key idea is the importance of reusability of the objects specified. An object should not only *work*, but should also be reusable. Programmers have their own methods for making classes reusable, but revision and reworking are always necessary. Each object must be made as

functionally independent of others as possible; this is hard to do and a difficult skill to acquire.

A third important idea is that the set of classes should provide a simulation of the system. This provides a concreteness that makes object-oriented systems easier to think about. A further key issue is the use of an existing library of classes; programmers browse through the library hierarchy to find existing classes that may be reused as they are or with some refinement.

Object-based design and object based programming consists of making objects and adding functionality to them. It is easy to confuse the two. The difference lies in the level of detail provided for the methods and in the completeness of the resulting collection (in the case of object-oriented languages, the completeness of the resulting hierarchy of classes); designers often implement only a few typical objects from which the rest of the design will follow. We can conclude from the above that object-based design is quite different from traditional design. Furthermore, Object-based is intrinsically more complex than the methodologies traditionally used for procedure oriented languages. The goal of the designer is to develop a simulation of the system to be implemented using objects as an abstraction, with the constraint that the specification of these objects must be as generic and independent as possible to guarantee maximal reusability.

There are several design principles that essential for Object-based design. They include ego-less programming, as object-based designers are perceived as library enhancers, and exploration of the design domain and the problem space, as object-based designers must avoid overspecialization of objects to further reusability. These principles must be at the center of any object-based methodology and such a methodology must be focussed on conceptual prototyping, the key ingredient missing from traditional approaches to design. Furthermore object-based design as an activity must be supported with interactive tools for iterative design revision and modification.

A Methodology for Object based Design

We will assume as starting point for the design of any software system to be a set of functional specifications. With an object-based language as implementation target, the designer must transform these functional specifications into a set of objects with their own functionality, state and attributes. In a nutshell, we can say that the design is a mapping from the functional specifications to a set of abstract objects. This mapping is at a considerably high level of abstraction where the functional specifications are partitioned among a well defined, highly independent, self-contained collection of units of abstraction and reuse.

The design methodology we propose is a hybrid between top-down design and data-driven design. To wit, we start with a functional decomposition as in top-down design. Having realized the top-level part of the functional decomposition, we proceed by identifying the data manipulated by each functional part. For each functional part we determine the data which is receiving the action specified or playing the acting role in the operation. By doing this we will get a handle in the identification of the necessary objects which will support the functionality at hand.

These data identification will produce the top-level objects of our design, corresponding to the top-level design. At this point we completely specify each of these objects by providing a specification of attributes and functionality.

We then proceed in an iterative way by providing a decomposition of these top-level objects into sub-objects based on the attribute specification and the functional specification of each object. For each attribute, using the attribute specifications, proceed to determine whether we are dealing with a simple primitive object or type that is supported directly by the language or if it is a new object in its own right, which we will then proceed to specify. For each operation of the functional specification of the object, we proceed to decompose it and apply the same method to determine possible new objects from each of its functional components.

Specification of the functional components of an object is comprised of two parts: the external functional interface, which is the set of operations available for interaction with other objects, and the internal functionality of the object, which consists of all the operations needed to create, update, and look up its internal state. Thus for the specification of a given object, we must identify a set of operations and then in turn a set of objects needed by the given object to support the required functionality.

We use functional specification to determine objects which in turn will produce subobjects to support them. This is a mechanism that can be used to determine the existence of objects, and whose existence is driven by the functional behavior being sought.

As we stated before, object-based design is an iterative process of object design whose activities include designing, implementing, and testing at the object level; each iteration has as goal the refinement and correctness of an object. These type of activities are recognized in [5] under the name *round-trip gestalt design*, and whose essence is the evolutionary process of object design to a point where the object its seen to have the right abstraction and semantics.

During the design process of an object the designer must recognize whether the object, as currently specified, is an library object, a refinement of a library object, or a new object for the library. We must realize that object-based design reduces to abstract object specification and to object refinement of an existing library object. With a rich and mature library, the latter is most common. The interaction of the designer with the library is critical in the design process; thus software tools must be supplied to support this activity.

Object based design is an evolutionary process both through the design process and through the life of the object in the library of objects. Experience dictates that object-based design follows the following schedule of activities :

- Object Identification
- Object Specification
- Object evaluation
- Object implementation

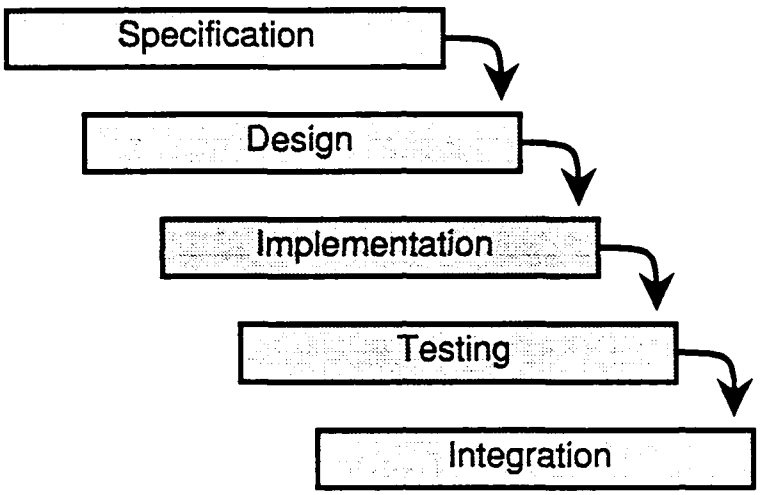
Functional specification is one methodology proposed here to aid in object identification. There are a number of other guidelines for object identification which have been proposed by several researchers in the area [3,5,7]; these guidelines list sources for object identification such as tangible things, roles, events, interactions, concepts, structures, etc. Object specification includes the semantical description of the external behavior of the object (interface, or protocol) and the list of attributes which will constitute the object's state. It is here that iteration of specification and implementation plays a crucial role. The goal is not only to produce an object, but a simple reusable one. Object evaluation is required as the resulting object must be evaluated with respect to the collection of objects which have been designed and with respect to the existing library of objects. Among the goals of this activity include the recognition of relationships among existing objects to simplify design and increase reusability. Object implementation will exploit object encapsulation by designing and implementing the interface in such a way that any chosen data representation is not reflected in the interface or affected by it in subsequent revisions. Software engineering provides guidelines for object implementation using modules, via the metrics of cohesion and coupling.

The process must have its own correctness procedure to guarantee that the objects meet the functionality found at each level of functional decomposition. Therefore, the process is an iterative one between top-down functional functional decomposition and object design. Each level of functional decomposition from the top-down design is embedded into the state and functional description of a set of objects determined by the given functional specification at each level. Following the top-down design methodology, we perform functional decomposition and show that at each level the functional decomposition is correct and complete. Then correctness check at object level must show that the functionality of the objects found subsumes the functional specifications at each level. In other words, we must show that the objects as specified do provide the functionality behavior as specified at a given design level.

For object-based design the goodness of the design is not only measured by the functional correctness of the resulting design, comprised by a set of objects; it must be measured by the reusability of the objects identified as well. We must bear in mind that while software reuse is in principle an effective method for the addressing of the software crisis, software reusability does not happen, it must be planned and designed for; in particular programming languages must have been designed to effectively support and promote software reuse. Designing for reusability is a hard skill to acquire and teach; experience can certainly make a difference in this software design aspect. See [9]. The iteration process of object specification and object implementation aids in the search for a design at a sufficient level of abstraction and reusability.

Life-cycle for object base languages.

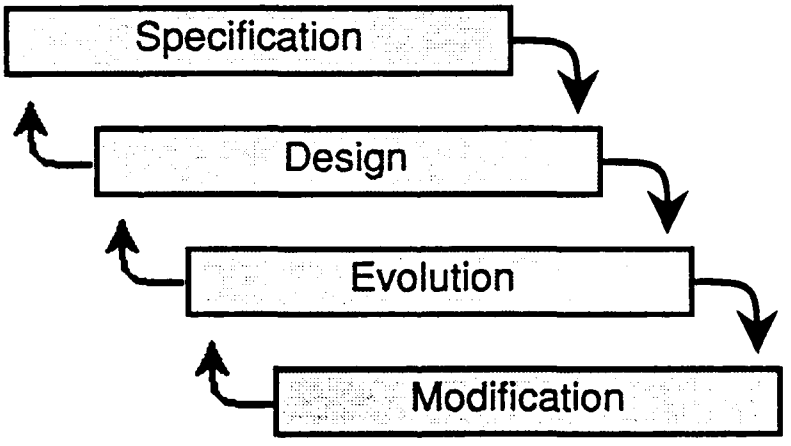
Modern programming languages such as Ada were designed by the pressing need to provide support tools for the development of maintainable, and extensible software systems. As we have seen earlier, these languages require a design methodology that makes use of their abstraction and encapsulation features. Furthermore, the entire process of software development cycle ought to be revised. We are all too familiar with the traditional waterfall software development cycle.



Waterfall Software Life-cycle for Structured based Languages

This life cycle has been strongly criticized as it does not adequately support the concerns of the development of highly evolving software systems. Boehm [2] has proposed the so-called spiral design which creates a risk-driven approach to the software process, rather than a strictly specification-driven or prototype-driven process.

The key characteristic that a life-cycle must support is the software evolution through an incremental, iterative design and revision process. The often heard phrase within the object-oriented developers, *Design a little, Implement a little, Test a little*, comes to mind. It embodies the essence of both object-based design and software evolution. In this view, the software life cycle for object-based languages can be picture as



Software Life-cycle for Object-based Languages

where the evolution activity deals with additions, refinements and revisions of the library of abstract objects.

Support Tools

Object-based design methodology as presented, and object-based implementation require software tools for the browsing of existing libraries of objects. These tools are already found in several programming environments for object-oriented languages such as Smalltalk, Objective-C, and Trellis/Owl among others. These tools, while helpful, increase dramatically the learning curve for a beginner designer/programmer using the object-based model [17]. Currently, we are designing a software design and development environment for object-based design having as design goal the production of a tool with the least amount of obstruction in the task at hand, and which will support the design, evaluation, implementation process of objects as described above. A component of this software design environment will be the implementation of a visual tool to aid in object design. This tool is in the spirit of the CRC-cards (Class, Responsibility, Collaboration), as presented in [1], which have been used in the teaching of specification of objects for object-oriented languages. CRC-cards are presented to be used as both a visual and thinking aid for the teaching of object-oriented thinking. They are used to define objects by name, responsibilities and collaborators; these last two, represent the functionality of the object at hand. The authors reported a high level of success of their use in teaching novice programmers the concepts of objects and in introducing programmers experienced with other paradigms to complex designs.

Conclusion

Current software requirements and volume mandate further revision of existing software development methodologies and their supporting tools. Modern modular programming languages designed to address these issues must be complemented with an appropriate software methodology akin to the new paradigm embodied within software modularity. Object-based design raises the unit of abstraction in design and by doing this it provides a methodology to exploit the high level features for software implementation such as encapsulation and generics. Functional decomposition, via top-down design, is a design methodology to guide the identification and design of objects, which should be seen as basic units of abstraction and software modelling.

Bibliography

[1]Beck, K, Cunningham W. *A laboratory for teaching Object Oriented Thinking.* Proc. of the Fourth Conference on Object Oriented Programming Systems, Languages and Applications. pp. 1-6, 1989

[2]Boehm, b. *A Spiral Model of Software Engineering Development and Enhancement.* Software Engineering Notes. Vol. 11 (4) , 1986

[3]Booch, G. *Software Engineering with Ada.* The Benjamin/Cummings Pub. Co. 1983

[4]Booch, G. *Software Components with Ada.* The Benjamin/Cummings Pub. Co. 1987

[5]Booch, G. *Object Oriented Design with Applications.* The Benjamin/Cummings Pub. Co. 1990

[6]Coad, P. Yourdon, E. Object-Oriented Analysis. Yourdon Press Computing Series..1990
[7]Cox, J.B. *Object Oriented Programming . An evolutionary Approach.* Addison Wesley, 1986
[8]Goldberg A, Robson D. *Smalltalk-80: The language and its Implementation.* Addison-Wesley, 1983
[9]Johnson, R., Foote B. *Designing Reusable Classes.* Journal of Object Oriented Programming. June/July 1988.
[10]LaLonde, W. R. *Designing Families of Data Types Using Examplars.* ACM TOPLAS. April 1989. Vol 11, No 2. pp 212-248
[11]Lalonde W.R. Panel : *Experiences with Reusability.* Chair by Kent Beck. Proc. of the Third Conference on Object Oriented Programming Systems, Languages and Applications. pp. 372-376. 1988
[12]Lieberherr K. J, Riel A. J. *Contribution to Teaching Object-Oriented Design and Programming.* Proc. of the Fourth Conference on Object Oriented Programming Systems, Languages and Applications. pp. 11-22, 1989
[13]Knudsen J.L, Madsen O. L. *Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages.* In European Conference on Object Oriented Programming. S. Gjessing and K, Nygaard, editors. pp 21-40. Springer-Verlag. 1988
[14]Meyer, B. *Reusability : The case for Object Oriented Design.* in [2] Vol II. pp. 1-34
[15]Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1988
[16]Nino, J. *Object Oriented Models for Software Reuse.* Proceedings of the IEEE SouthEastcom. New Orleans, april 2-5, 1990.
[17]O'Shea, T. *Panel : The Learnability of Object-Oriented Programming Systems.* Proc. of the First Conference on Object Oriented Programming Systems, Languages and Applications. Sigplan Notices, vol 21, no 11. pp. 502-504, 1986
[18]Shaffert C, Cooper T, Bullis B, Kilian M, Wilpolt. *An Introduction to Trellis/Owl.* Proc. of the First Conference on Object Oriented Programming Systems, Languages and Applications. Sigplan Notices, vol 21, no 11. pp. 9-16, 1986
[19]Wegner, P. *Dimensions of Object-based Language Design.* Proc. of the Second Conference on Object Oriented Programming Systems, Languages and Applications. 1987
[20]Wirfs-Brock R, Wilkerson B. *Object-Oriented Design: A responsibility driven Approach.* Proc. of the Fourth Conference on Object Oriented Programming Systems, Languages and Applications. pp. 71-75, 1989

This Page Left Blank Intentionally

Implementation of Artificial Intelligence Systems in Ada

Sumitra M. Reddy
Department of Computer Science
West Virginia University
Morgantown, WV 26506

1 Introduction

Artificial Intelligence (AI) systems are normally implemented in LISP, Prolog or in specialized languages, such as OPS5, KEE, LASER [1]. When performance became an issue, conventional languages (especially the C language) have found their way into AI programming. At the same time, Ada has become a major programming language for building large software systems. Also, many large systems have begun exploiting AI programming techniques for problem solving. These observations motivated us in West Virginia University to teach a special topics graduate course (started in Spring 1988 and taught every spring semester since then) using Ada as the implementation vehicle for AI techniques.

The primary goal of the course was to train students who can opportunistically exploit special features of Ada in elegant and robust implementation of well known AI paradigms.

Emphasis of the course was on actual implementation of AI systems using special features of the Ada language such as *packages, private data types, tasks* and *exception handling.* Specifically, Ada was used as "Ada" as opposed to translating an existing system into Ada. The course was targeted at advanced graduate students in Computer Science with experience in implementing large programming projects. The students, in the last three semesters the course was taught, were proficient in C. A few were also familiar with AI techniques and the Ada language.

In the rest of the paper, we will briefly outline the course syllabus, Ada features that have been exploited in implementing AI concepts and the description of the semester projects.

2 Course Syllabus

The syllabus was divided into two tracks. The first track was designed to provide sufficient working knowledge of Ada to the students, assuming that they had no previous experience in Ada. The second track consisted of lectures on major topics in Artificial Intelligence including a variety of programming paradigms. Because of its importance and relevancy to the assignments/projects, the concept of Ada packages was introduced at the very beginning of the course.

[1] Commercial systems

To familiarize the students with the basics of AI programming, the following paradigms were introduced.

- Object-oriented programming.
- Data-driven programming using *demons*.
- Rule-based programming.

  The following major AI techniques were discussed in the class and subsequently some of these were assigned as tool-building projects to be used later in the semester for developing a mini-expert system.

  - Knowledge representation techniques using *semantic networks, frames, scripts* and *rules*.
  - *Inheritance* to reduce redundancy.
  - *Production systems* using problem solving rules and forward and backward chaining *inference* engines.
  - Uncertainty and probability for handing incomplete knowledge.
  - Natural language processing using grammar-based approaches.
  - *Blackboard* model for problem solving in a cooperative environment.

Discussion of the AI techniques was interspersed with various Ada topics and good software engineering principles. Particular emphasis was placed on *packages, information hiding, private data types, variant records, access types, tasks, overloaded subprograms*, separate compilation, and exception handling since these concepts presented the means to build robustness into implementation. *Generics* were also discussed for various possible applications, but these were not implemented due to time constraints.

3 Semester Project

Since the primary goal of the course was to train students who can incorporate AI techniques into Ada programs, they were assigned a semester long project which allowed them to solve a complete problem using the tools developed as a part of the project. The project consisted of two parts:

1) Development of AI tools comprising of a knowledge representation package to build knowledge bases containing both facts and rules, and an *inference engine* that operated on a knowledge base.

2) Building a mini-expert system as an application of the AI techniques to solve a complete problem using the tools developed in the first part of the project.

In the next two sections, we describe the AI tools developed by the students. Applications are described in the third section.

3.1 Knowledge Representation Package

The students developed a hierarchical object-oriented knowledge representation package using the *frame* style approach (in the *object-slot-value* format) that allowed relations and inheritance.

The purpose of this package was to enable the users to **dynamically** build, peruse and manipulate knowledge bases. Multiple values as well as different types of values (for example, string, integer, float etc.) were allowed to be placed in any slot of an object.

The package specification consisted of a set of operations (subprograms) such as *create, delete, display, relate* etc. on object/slot/value wherever they are applicable. The package body contained implementations of these subprograms and also support operations such as searching for an object, slot or value before *create, delete* or *relate* operations could take place. To foster incremental development of the package, *body stubs* were used in the package body for separate compilation.

Overloaded subprograms were used to perform similar operations on an entity (an entity may be an object, slot or a value). For example, to create an object, only the name of the object (hence only one parameter) was required. To create a slot, two parameters: the name of an object and the name of the slot were required. To create a value, three parameters were required: the object, slot and the value. The overloading feature of Ada allowed implementation of the three versions of *create* to accomplish the creation of the desired entity. Following are examples of the *calls* to the overloaded subprograms of *create* in *pretty print* form:

```
create(dan)
create(dan rank)
create(dan rank graduate)
```

The *relate* operator was used to connect any two existing objects in the knowledge base using either the predefined *isa* relationship or using any user-defined relationship. Only the *isa* relationship allowed for inheritance. Inheritance mechanism was implemented to enable retrieval of the *values* from the *super objects* in the inheritance hierarchy if no values are found locally in the slot of an object.

Before relating any two objects using a user-defined relationship (i.e. other than *isa*), the relation pairs, such as *master-pet, parent-child* were first created using the *create-rel* facility.
An example of an object (in *pretty print* form) is given below.

```
{{ dan
       isa: student
       rank: graduate
       age: 22

 }}
```

In the above example, *dan* is the object with the slots *rank* and *age*, and is a member of the class *student* (another object). The inheritance mechanism allows *dan* to inherit *values* from the *super objects* in the inheritance hierarchy.

The same package also facilitated creation of problem solving knowledge in the form *rules*. These will be discussed in the next section.

3.2 Inference Engine

The students implemented a generic (not to be confused with Ada *generics*), domain-independent forward chaining inference engine to process user-defined problem solving knowledge expressed in terms of *rules*. The purpose of the inference engine is to apply problem solving rules to the domain knowledge created by the users of the knowledge representation package.

The rules, which are also *objects* similar to other objects containing declarative facts in the knowledge base, were created by the users in a *prescribed* format, each with an *if* slot containing a set of condition clauses as the *value*, and a *then* slot containing the "actions" to be taken if the conditions were satisfied.

The conditions in the *if* slot were required to be expressed in *object-slot-operator-value* format. The same facilities implemented in the knowledge representation package (the first AI tool) were used to create these *rule objects*. Variables, signaled by a special character ?, were also allowed in the rules. Example of a rule object that uses a variable is shown below.

```
{{ rule2
     isa: rule
     comment: '' Feed dog-food to a samoyed
                 if it is hungry''
     if:      ?x isa = dog
              ?x type = samoyed
              ?x food = none
              ?x hungry = yes

     then:    create(?x food dog-food)
              create(?x hungry = no)
}}
```

The object *rule2* contains the slots *comment*, *if* and *then*, and is a member of the class *rule*. Every rule was created as a member of the *rule* class. This designated class of objects were specially treated (interpreted) by the engine to solve problems.

The value of the *if* slot has four sets of conditions ANDed together. The *comment* slot contains the description of the rule in English. Each condition is expressed in terms of four entities: <object> <slot> <operator> <value>.

In the first condition, *x*, *isa*, = and *dog* are the object, slot, operator and the value respectively. It states that "if there is an object (not known yet) that is a member of the *dog* class". The second condition is interpreted as "if the object x (matched in the first condition, hence known) has a slot called *type* with a value *samoyed* ". The other two conditions can be interpreted in the same way.

The (?) prefix for the variable x signals the engine to perform matching against the objects in the knowledge base. The variable x is then bound to the object that is matched first. The matched object is used in the second condition onward. In case of a match failure in any of the subsequent conditions, the engine picks the next matched object and the conditions are evaluated again. To simplify the system, match variables were not allowed in the place of an operator; these were explicitly named, such as "=", ">" or "<".

The *master* task of the inference engine dynamically spawned the required number of Ada tasks (same as the number of rules in the knowledge base, hence varied from application to application), and assigned each task to process/interpret a specific rule. As a result, the conditional parts (the *if* slot) of all the rules were "concurrently" [2] processed by the designated task to determine its eligibility to be *fired*. The rules are normally processed sequentially in the systems implemented in other languages.

Since each task performed the same job of matching the facts in the knowledge base against the conditions, a *task type* was used to implement this feature. Using an *access type*, the required number of tasks were dynamically created as needed by the problem. ¿From the *conflict set* of eligible rules, the master task selected one rule depending on the conflict resolution criteria; the actions specified in the *then* part of the selected rule was then executed. The engine repeated the match-select-fire cycle until no rule was eligible or an explicit HALT action was encountered.

3.3 Applications

To demonstrate the capabilities of the inference engine, the students performed syntactic analysis of simple declarative sentences using context-free grammar rules, such as S → NP VP ; NP → DET ADJ N ; NP → PRN ; VP → V ADV ; VP → V PP ; etc. Based on the set of grammar rules, an input sentence was either accepted or rejected by the engine.

In addition to the syntactic analysis problem, students also implemented a mini-expert system using the tools developed in the class. Typical problems, solved by the mini-expert systems built by the students, were scheduling of classes, assigning a classroom, getting a car from the university's motor pool, and various other university-related activities which can be "solved" using rule-based techniques.

[2] True concurrency is realized only on systems where multitasking is done using multiprocessors

Since the engine was designed to be generic, it was not hard-coded for any specific problem. Problem specific information, described by relevant *facts* and *rules*, were input by the user to create the knowledge base.

4 Conclusions

Even though the course had the ambitious goal of introducing AI techniques and the features of Ada, we found it feasible to rapidly cover both topics because of the advanced preparation of the students. This could not have been feasible if this was offered to beginning students.

Ada package structure was found to be very natural in implementing the knowledge representation system with controlled visibility. All the data structures were hidden from the users of the package; only the functions to develop and manipulate the knowledge base were provided in the package specifications. Exception handlers were developed to print appropriate error messages.

In addition to the inference engine, another possible use of the Ada tasks discussed was the implementation of a *blackboard* (the master task) that uses *cooperative*, multiple knowledge sources (hence multiple Ada tasks that can operate concurrently) communicating through the blackboard. An extended version of this course spanning two semesters could include implementation of the blackboard concept and other AI techniques.

The project required extensive manipulation of symbols. Therefore, two different string handling packages for processing variable strings were provided at the beginning of the project. While most of the students selected the package of their choice, some wrote their own string package.

Since Ada does not provide a *subprogram* type, different (and less straightforward) ways of implementing the data-driven programming using *demons* were discussed, but not actually implemented.

Postscript: A number of students who went on to work on different AI projects following the class remarked that their Ada experience was very beneficial as well as influential in designing software implemented even in languages other than Ada.

STANFINS Redesign Subsystem II: Developing A Large MIS Application Using Ada

by Richard M. Somers
Project Manager
Computer Sciences Corporation
Applied Technology Division
P.O. Box 16008
Indianapolis, IN 46216

Abstract

STANFINS Redesign Subsystem II (SRD-II) is a very large MIS application (1.8 million lines of code) that is being written in Ada. As a result of its being both very large and an MIS application, many interesting challenges were encountered during its development.

Part 1 of this paper contains a brief overview of SRD-II including a discussion of its functional requirements and types of processes.

Part 2 of this paper discusses several of the many challenges that have been encountered and the solutions that were developed. Included in these are: COBOL to Ada transition, Rational development tools, Decimal Arithmetic Package, Host Character Package, operating environment interfaces, Rational to MVS transition, and testing on different compilers.

Part 3 of this paper discusses the various methods of training that have been utilized. Included in these are: lecture/ hands-on, computer aided instruction, newsletters, self instruction, and walk throughs.

Part 4 of this paper summarizes conclusions regarding training. Included are: COBOL language/MIS background vs. PASCAL-related language/non-MIS background, effectiveness of the various training methods, and recommendations.

Overview of SRD-II

The contract for STANFINS (Standard Finance System) Redesign Subsystem II (SRD-II) was awarded to Computer Sciences Corporation (CSC) in September 1986. Like the majority of Management Information System (MIS) applications, SRD-II was directed to be written in COBOL.

STANFINS is the U.S. Army's field-level accounting system for general funds servicing posts, camps, and stations. STANFINS Redesign is an interactive system that replaces the current STANFINS batch system. It is composed of two (2) subsystems: Subsystem I and Subsystem II.

Subsystem I is divided into Travel, Accounts Payable, and Disbursing and was developed independently.

Subsystem II is divided into the following ten (10) components:

1. Funds Receipt

   This component distributes program and budget guidance, permits adjustments of program requirements based on actual guidance received, and calculates dollar and percentage deviations between program and actual.

2. Funds Status

   This component provides automated support and general ledger control for the expenditure stages of accounting (commitment, obligation, accrual, and disbursement).

3. Accounts Receivable

   This component establishes accounts receivables, records receivables, produces original and follow-up bills, and processes collections.

4. General Ledger

   This component generates trial balances monthly and annually, generates cash reconciliation reports, and provides adjustment capabilities to several installation reports.

5. Other Funds

   This component separates and forwards various Transactions for Others (TFO), Transactions by Others (TBO), Cross Disbursements (XD), and Miscellaneous

transactions.

6. Fixed Assets

   This component accounts for the real and personal property owned by the U.S. Army. It provides a method for recording, capitalizing, and depreciating real and personal property.

7. Cost Accounting Standards

   This component establishes accurate labor performance standards, summarizes and interprets labor performance, and provides trend analysis.

8. Cost

   This component calculates labor cost, accrues leave and benefits for civilian employees, calculates equipment usage cost, applies overhead, and stores total cost by job order, cost center, and element of resource.

9. Tables Maintenance

   This component provides the capability to interactively maintain data used by the system to edit and validate other data.

10. System Support

    This component controls access to menus and screens, allows maintenance of programs, and allows control of batch jobs.

By any measure SRD-II is a very large system. The following statistics may provide an example of how large the system is:

Lines of Code: 1.8 million (approximately)
Programs: 528
Screens: 381
Records: 175
Input Interface Files: 39
Output Interface Files: 26
Reports: 137

The types of processing done by these programs can be summarized by the following five (5) processes:

1. Auxiliary Support

   This process provides access control and system security, database administrator capabilities, system

maintenance capabilities, end of period processing, and batch job scheduling.

2. Batch Data Store Maintenance

   This process receives files from interfacing systems, edits them, and updates system files.

3. Batch Information Generation

   This process produces hardcopy reports too large to be run interactively and generates interface files for other systems.

4. Interactive Data Store Maintenance

   This process enables the end user to interactively update system files.

5. Interactive Information Generation

   This process interactively displays small reports generated from system files.

The net requirement was to develop an interactive system that received data from other systems, generated data to other systems, retrieved/updated large volumes of records, and produced hardcopy reports. This type of application was perfect for the traditional COBOL language and staffing was done accordingly.

Challenges and Solutions

The project was faced with numerous challenges none of which were insurmountable. Several of these challenges and the solutions that were developed are listed below.

1. COBOL to ADA Transition

   The first major challenge occurred in May 1987 when the contract was modified to require Ada as the programming language instead of COBOL. Since the application was MIS, the preferred solution was to locate and hire experienced Ada/MIS programmers and analysts. It became readily apparent that these people did not exist. Two choices then remained: hire Ada programmers and teach them MIS fundamentals or teach Ada to the COBOL/MIS people already on staff. The second choice was selected as the most desirable from both time and cost considerations, and this

decision proved to be correct. The COBOL/MIS background proved to be invaluable in the development of the project.

2. Rational Development Tools

Due to the enormous amount of source code to be generated, it soon became apparent that a special development tool was required. Numerous compilers were looked at and the Rational R1000 was finally selected. The Rational R1000 offered excellent editing capabilities, full debugging capabilities, testing facilities, and a configuration management system. Initially four (4) processors were acquired in the summer of 1987, but currently six (6) are being used due to the volume of code.

3. Decimal Arithmetic

The accuracy of floating point and fixed point types provided by Ada is compiler-dependent. The target compiler used in the IBM/MVS environment can only provide 8 to 9 positions of accuracy. It was determined that 14 positions of accuracy were necessary due to the kinds, size, and volumes of numbers used by the system. As a result, CSC developed the Decimal Arithmetic Package that has been used to enter, retrieve, and store decimal values with the larger required positions of accuracy.

4. Host Character Package

Ada's standard string type is based on the ASCII character format. The target environment, however, has an EBCDIC character format. CSC developed a Host Character Package which contains a new string type and corresponding functions based on the EBCDIC character format.

5. Interfaces

Interfaces were developed to provide access to operating environment functions. Included in these were a CICS interface which provides I/O capability, a Datacom/DB interface which provides database retrieval and storage capability, and an operating system interface that provides access to the target environment operating system.

6. Rational Environment to Target Environment Transition

Code was developed and tested on the Rational R1000 and then ported to the target environment. It was

expected that differences would be detected between the environments due to fact that one was a development environment and other was a target environment. It was hoped, however, that the differences would be minimal, and this turned out to be the case. Some of the differences were the result of compiler deficiencies and others the result of architecture limitations. A few of the differences that were discovered are listed below.

Many subunits were initially created to allow minimum recompilation if changes had to be made. The Rational R1000 handled these very efficiently. The target environment could not due to a deficiency in the target builder. As a result, subunits were brought inline prior to porting to the target environment.

Many early generics were nested. That is, generics had instantiations of other generics. Again, the Rational R1000 handled these very efficiently. The target environment, however, would not compile nested generics. Those generics and the dependent generic instantiations had to be rewritten.

Once the characteristics of the target environment were determined, the code written on the Rational R1000 was adjusted accordingly.

7. Testing on Different Environments

All programs were tested on the Rational R1000 prior to porting to the target environment. As mentioned above, interfaces were created to take advantage of the target operating system. The Rational R1000 had its own set of interfaces for its own operating system. As a result, it was necessary for to develop software to emulate the target environment on the Rational R1000. This provided a greater degree of confidence when testing actually began on the target environment.

Many of these challenges are common to any software development project and should not be considered as unique to Ada/MIS applications. Often the development environment and the target environment are different and present similar problems. The one (1) challenge that is unique is the transition from COBOL to Ada. Training was essential to accomplish this transition.

Training Methodologies

Once the decision was made to implement the project using Ada, many efforts were made to provide meaningful and effective Ada and Ada-related training. This process began long before the actual coding phase and was offered to all employees regardless of their assignments on the project.

1. Lecture/Hands On

   Initial training was provided by CSC and CSC-contracted employees in a lecture/hands-on format. This was accomplished by an 80-hour course presented 8-hours per day over a two-week interval. It included a review of object oriented design concepts, exposure to Ada concepts, and hands-on training using the Rational R1000. Initial employee response was very poor for several reasons. The instructors had only about two (2) months of Ada training themselves prior to teaching the course and were able to provide little more than superficial knowledge about Ada. The instructors were technical personnel, not professional instructors, and were not able to convey information in an efficient manner. The only Ada training material available was ANSI/MIL-STD-1815A-1983 and it was much too technical to be used effectively. The employees were, by and large, not familiar with block structure languages and resisted accepting Ada. Also, the employees attending the course were not yet using Ada on the project, so there were no work-related practical applications where skills learned in class could be used.

   These deficiencies were overcome by making several changes to the course structure. Richard E. Bolz was hired as a consultant to provide professional Ada training. This strategy was extremely effective in both the short term and long term. In the short term it provided the much needed professional instructor that was both knowledgeable and able to present concepts efficiently. In the long term it allowed CSC to develop instructors that could learn from Mr. Bolz and fill the void created when he was not available. He permitted CSC to use and reproduce the materials he developed. An introductory course was developed using his Software Engineering with Ada materials. In-class exercises were added to provide a method to emphasize the major objectives of the course. A copy of the course outline that is presently being used for this introductory course is included as Appendix A, and a copy of the in-class exercises is included as Appendix B. An intermediate course was

developed using his AdaVenture materials. Advanced courses were developed to teach advanced Ada concepts such as generics and tasking. Ada-related training, primarily Rational R1000 training, was still conducted by CSC instructors, but was more effective because the new instructors were more frofessional. When the time came to actually write code, refresher courses were offered to reacquaint employees with Ada and Ada-related skills.

2. Computer Aided Instruction

To supplement the formal lecture/hands-on training, computer aided instruction (CAI) was offered to employees. CAI included both a tutorial and practical exercises. It allowed employees to progress at their own pace and provided the capability to review material that was unclear. It also recorded employee progress for management review. Initial employee reaction was again unfavorable primarily because there was not a perceived need to know Ada since it was not then being used and a reluctance to learn Ada since its syntax and structure was viewed as threatening.

The reluctance was overcome once the formal training improved and the project moved closer to implementation. Employees discovered that they could learn Ada and that motivated many to want to learn. CAI was successful at letting them experiment with the language.

3. Newsletters

Newsletters were initiated as a way to provide regular informal training concerning Ada and Ada-related topics. Each issue concentrated on a specific topic and contained an exercise at the end to reinforce the content of the issue. The early issues were devoted to the Rational R1000 since it was necessary for employees to become acquainted with it before writing code. Later issues concentrated on specific Ada topics. The newsletters were well received because they were written in layman's terms rather than the technical language of ANSI/MIL-STD-1815A-1983. It also had the unintended benefit of forcing the author to research material in preparation for each issue and thereby become more knowledgeable himself.

4. Self Instruction

Many employees purchased books about Ada to supplement training. While there were many books available, most were technically oriented and assumed a level of

knowledge beyond that of the introductory student. The book that was found to be the most helpful was Software Engineering with Ada by Grady Booch. This book has a very readable style and is ideal for the introductory student. It contains numerous examples and provides many exercises. Unlike formal training or CAI, it could be taken home by the employee overnight or over the weekend.

5. Walk Throughs

   Once Ada code was written, walk throughs were established to provide a way to present how code was written at the project level. The walk throughs allowed team members to observe different ways to write code. While each programmer had their own style, each could discover ways to improve their code by looking at the way others were writing code. This was very valuable initially because few programmers knew how to write Ada very well and no one knew how to write an MIS application using Ada. The walk throughs provided an informal forum for discussing problems and sharing solutions, and gradually helped develop the confidence of each programmer.

Conclusions

Following are some conclusions that have resulted from the experience of developing SRD-II:

1. COBOL Language/MIS Background vs. Pascal-Related Language/non-MIS Background

   Programmers with Pascal-related language backgrounds generally accepted Ada and developed a proficiency with Ada much quicker than those with COBOL language backgrounds. This was expected since Ada is based on the Pascal language.

   Programmers with COBOL language backgrounds tended to resist Ada because from their experience MIS applications were written in COBOL. They also had difficulty understanding many Ada concepts like strong typing, package structure, local variables, block structure, and parameter passing. The greater the resistance or frustration, the longer it took for them to accept and learn Ada. This does not mean that programmers with COBOL language/MIS backgrounds can not learn Ada or that they should be discouraged from doing so. One of the most proficient programmers on

the project is a former COBOL programmer who has fully accepted Ada.

Programmers with COBOL language/MIS backgrounds can provide the much needed MIS experience that programmers with Pascal-related backgrounds often do not have. Being able to write Ada code is not enough. One must understand MIS applications and be able to analyze requirements to write Ada/MIS code well. One of the most respected analysts on the project is a former COBOL programmer with 20+ years of MIS experience.

People from both backgrounds will be necessary until people from the single Ada/MIS background become available. This project has developed the first generation of these people.

2. Effectiveness of the Various Training Methodologies

By far the most effective training methodology was lecture/hands-on as it provided the introductory exposure to both Ada and the Rational R1000. It allowed employees to learn both Ada concepts and the Rational R1000 in a classroom environment where questions could be asked and answered without the employee feeling intimidated. The immediate feedback offered by an instructor can often answer a question or solve a problem for an employee that CAI or a book could not readily answer. Employees also benefit from the questions that others ask.

The next most effective training methodology was the walk through as it provided the employee with a peer group with whom he/she could exchange information on the actual implementation of Ada. This dynamic atmosphere provides valuable experience and confidence.

CAI was effective only if the employee was motivated to pursue it. It is a passive environment that responds only to questions included in the tutorial. Questions the student may have that are not included in the tutorial will not get answered. It also does not provide the interaction of a group setting. The employee does not get the benefit of the answer to a question someone else asks.

The newsletters provided reinforcement of Ada and Ada-related information at regular intervals in a informal format. Its value was as a supplement to formal training and as reference material.

The least effective training methodology was self-

instruction primarily because the format of most books is not oriented to training. Even the best of books do not and can not answer all questions. The employee will often begin a book with good intentions only to lay it aside when it becomes boring, confusing, or is not project related.

3. Recommendations

While programmers with an Ada or Pascal-related language background will probably be able to write Ada code very quickly, do not discount the programmer with a COBOL/MIS background. The majority of the development lifecycle is application dependent and language independent. Thus, the MIS experience these people can offer to a project is every bit as valuable as the ability to write Ada code.

Start Ada training near the time when it will be used. To start earlier is inefficient because employees will be busy with current assignments and will not retain the knowledge or skills taught. They are will also be more receptive to learning skills that can be immediately applied.

Ensure that instructors are knowledgeable and professional. If they are, the courses will be more efficient and the employees more receptive to the information being taught. If necessary, hire consultants to provide initial training. They can provide that much needed initial exposure to Ada. Consultants can not always be there, however, so it is very important for a project to eventually develop its own instructors. They provide continuity and a visible link between the project and Ada.

Be open and candid with employees on how a project is progressing. This can be done with newsletters or with short briefings, but it is important that communication be maintained. Share the successes of the project so employees can be informed about what progress is being made. It builds morale. It is just as important, however, to share the setbacks. Rumors have a way of pervading a project. The effect of rumors can be minimized if setbacks are acknowledged and solutions explained. It builds confidence. Good communication can bind the employees on a project together.

SRD-II has demonstrated that a large MIS application can be developed using Ada. Other large MIS applications can now be considered because SRD-II has paved the way.

Appendix A

Introduction to Ada

Course Outline

A. Software Engineering
   1. Goals
   2. Principles
   3. Object Oriented Design

B. Overview of the Ada Language
   1. ANSI/MIL-STD-1815A-1983
   2. Introduction

C. Ada Syntax
   1. Character Set
   2. Lexical Units
   3. Reserved Words
   4. Delimiters

D. Rational Fundamentals
   1. Basic Mechanisms
   2. Ada Program Creation
   3. Ada Program Modification
   4. Additional Topics

E. Types & Subtypes
   1. Scalar
      a. Discrete
         1. Integer
         2. Enumerated
      b. Real
         1. Fixed
         2. Floating Point
   2. Composite
      a. Array
      b. Record
   3. Private
   4. Access
   5. Task

F. Program Units
   1. Packages
   2. Procedures
   3. Functions

G. Control Structures
   1. IF Statements
   2. Loop Statements
      a. Loop .. Exit
      b. For .. Loop

        c. While .. Loop
        d. In Reverse .. Loop
    3. Case Statements
    4. Block

H. Input / Output
    1. External Files
        a. SEQUENTIAL_IO (Generic)
        b. DIRECT_IO (Generic)
    2. Text Files
        a. TEXT_IO
        b. INTEGER_IO (Generic)
        c. FIXED_IO (Generic)
        d. FLOAT_IO (Generic)
        e. ENUMERATION_IO (Generic)
    3. IO_EXCEPTIONS

I. Visibility Rules
    1. Use Clause

J. Program Structure & Compilation Issues
    1. Context Clause (With)
    2. Compilation Units
    3. Subunits
    4. Compilation Dependencies
    5. Program Library

K. Exception Handling

L. In-Class Exercises
    1. Exercise 1
    2. Exercise 2
    3. Exercise 3

Appendix B

Introduction to Ada

Exercise 1

Objectives:

1. Gain an understanding of type and object declarations.
2. Gain an understanding of private types.
4. Gain an understanding of packages.
5. Gain an understanding of procedure declarations and calls.
6. Gain an understanding of Text_Io, Enumeration_Io, Integer_Io,and Sequential_Io.
7. Gain an understanding of control statements.
8. Gain an understanding of visibility rules.
9. Gain an understanding of compilation dependencies.

Requirements:

1. Login to Rational.

2. Create a world called Exercises in your home library.

3. Create an Ada package, Process_Employee_Record, that will contain the resources described in Requirements 4 - 7.

4. Declare a private record type, Employee_Record_Type, that includes the following fields:

    a. Last_Name up to 25 characters.
    b. First_Name up to 15 characters.
    c. Age between 0 and 125.
    d. Job_Title to be any of the following:

        (1) Programmer_Analyst
        (2) Senior_Analyst
        (3) Supervisor
        (4) Project_Manager
        (5) Manager
        (6) Other

5. Write a procedure Create_Record separate from the package that will accept user input and create a record of this type. To reduce keystrokes by the user and to reduce the chances of an entry being misspelled, the input of the Job_Title component should be of the multiple-choice variety that is validated by a case case statement. Also, investigate the use of attributes in a loop statement to provide a more efficient alternative to the case statement.

6. Write a procedure Print_Record separate from the package that will print the contents of this record to the screen in a readable form with labels.

   Example:

   ```
   Last Name: Doe
   First Name: John
   Age: 61
   Job Title: Manager
   ```

7. Write a procedure Create_File separate from the package that will read the contents of this record to a sequential file.

8. Write a driver procedure that will utilize the resources in package Process_Employee_Record to:

   a. call Create_Record
   b. call Print_Record
   c. call Create_File

Introduction to Ada

Exercise 2

Objectives:

1. Gain a further understanding of packages.
2. Gain a further understanding of private types.
3. Gain a further understanding of procedure declarations and calls.
4. Gain a further understanding of Text_Io, Enumeration_Io, Integer_Io, and Sequential_Io.
5. Gain a further understanding of visibility rules.
6. Gain a further understanding of compilation dependencies.

Requirements:

1. Login to Rational.

2. Go to the world called Exercises in your home library.

3. Modify the Ada package Process_Employee_Record so that it contains the resource described in requirement 4. Notice which units are affected by the change.

4. Write a procedure Print_File separate from this package that will print the contents of the sequential file created in Exercise 1 to the screen in a readable form with labels. Reverse the order in which the record from Exercise 1 is displayed so that First_Name is listed first and Last_Name is listed second.

   Example:

   ```
   First Name: John
   Last Name: Doe
   Age: 61
   Job Title: Manager
   ```

   Why must this procedure be contained in package Process_Employee_Record and not another arbitrary package?

5. Write a new driver procedure that will utilize the resouces in this package to call Print_File.

Introduction to Ada

Exercise 3

Objectives:

1. Gain a further understanding of Text_Io (Get_Line & Skip_Line).
2. Gain a further understanding of control statements.
3. Gain an understanding of block statements.
4. Gain an understanding of exceptions and exception handling.

Requirements:

1. Login to Rational.

2. Go to the world called Exercises in your home library.

3. Modify the Ada procedure Create_Record that is separate from package Process_Employee_Record so that it handles bad user input locally as described in Requirements 4 and 5. Notice which unit(s) is/are affected by the change.

4. Use procedures Get_Line and Skip_Line from package Text_Io so that regardless of user input for Last_Name or First_Name, the defined number of spaces (25 and 15 respectively) are allocated to each variable. If not enough characters are entered, fill in the required remainder with blank spaces. If too many characters are entered, truncate the excess and clear the buffer.

5. Enclose Age_Io.Get (Age) and Prompt_Io.Get (A_Prompt) within block statements so that in each case, local exception handlers prevent the program from being terminated when exceptions are raised due to invalid entry. Determine which exceptions are likely to be raised for each variable and develop local exception handlers to:

    a. Advise the user that the entry was invalid
    b. Clear the buffer
    c. Prompt the user for a new entry

    Enclose the block statement within a loop so the user only exits when a valid entry is made.

References

Bolz, Richard E., Software Engineering with Ada

Bolz, Richard E., AdaVenture

Booch, Grady, Software Engineering with Ada, Benjamin Cummings Publishing Company, Inc., Menlo Park, California, 1987.

Panel Session

Curriculum Trends in Software Engineering

Patricia Lawlis, Chair

Formal curricula in Software Engineering are just beginning to become widespread. This panel discussion addresses many of the issues involved with developing a curriculum in this relatively young discipline.

The panelists address the following topics:

- how software engineering curricula have evolved to present
- the present state of software engineering curricula
  - addressing undergraduate and graduate
- trends for the near future
  - addressing undergraduate and graduate
  - discussing accreditation issues
- should Ada be an integral part of a software engineering curriculum

Participants in the panel include:

LtCol Rick Gross	SAF/AQXA
Col Bill Richardson	US Air Force Academy
Dr. Paul Jorgensen	Research & Technology Institute of West Michigan
Dr. Gary Ford	Software Engineering Institute
Dr. Mike Feldman	George Washington University

Corporate Software Engineering Curriculum Needs

Richard R. Gross, Lieutenant Colonel, USAF
Office of the Assistant Secretary of the Air Force (Acquisition)
Washington, DC 20330-1000

My remarks come from the perspective of a *consumer*, rather than a producer, of software engineering education. The U.S. Air Force is typical of today's large technical industries in that our extrapolations to the year 2000 show demand for software materially exceeding our capability to produce or otherwise obtain it. Thus, we are critically dependent on a breakthrough that will alter this equation. Software engineering, of all the technologies now on the horizon, has the best potential to achieve such a breakthrough.

If software engineering is to alter the Air Force's supply/demand extrapolations significantly, however, two things must happen. First, classical engineering discipline must be applied to software. Such application, in turn, demands that we first learn *how* to do this; only then can we insert this discipline into our educational curricula. The second imperative, from the Air Force's perspective, is that we secure the services of an adequate proportion of these educated software engineers.

While colleges and universities are working on the first problem, we in the consumer sector can help them by pursuing the second. That is, in formulating and publicizing specific software engineering needs, we consumers can provide a visible incentive that will stimulate both the development of software engineering curricula and their insertion into educational programs. To this end, I anticipate that we in the Air Force will begin working soon with other governmental agencies and with the private sector to make our needs for educated software engineers public. As a first step, we are now working with the Software Engineering Institute to develop a definition of the term *software engineer* that we can use to determine how many software engineers we currently have and to forecast how many we will need.

We know of no other language that supports software engineering to the degree Ada does. Because it is so critical that we move swiftly and completely to software engineering, we have instituted a policy that Ada be used for all Air Force software unless a compelling lifecycle cost-benefit case is made for an exception. Naturally, then, we would prefer our software engineers to be facile in Ada and to have been educationally grounded in it.

Remarks at "Curriculum Trends in Software Engineering" Panel of the Fifth Annual Ada Software Engineering Education and Training (ASEET) Symposium, Alexandria, VA, August 15, 1990.

SOFTWARE ENGINEERING IN AN UNDERGRADUATE PROGRAM

William E. Richardson

United States Air Force Academy

Department of Computer Science

Over the past ten years, the precursors to software engineering curricula have been in development in both graduate and undergraduate programs around the world. In particular, graduate schools have made substantial headway in producing viable programs in software engineering. This progress stems from several sources, including the DoD sponsored Software Engineering Institute (SEI) graduate curriculum and innovative masters programs like the one developed by the Wang Institute. Additionally, the nature and immaturity of the software engineering discipline, along with the relatively simple requirements for developing a graduate program contributed to this progress in the graduate arena. There are now over a dozen recognized graduate programs in software engineering.

It appears that software engineering programs currently exist at the masters level because of historical evolution rather than absolute necessity and, indeed, some progress is also evident in undergraduate programs. However, there are several issues which have retarded the development of undergraduate software engineering curricula independent from computer science programs. The first problem is that much of the core of computer science is a necessary foundation for the understanding of software engineering. Because of this large common background, a complete break between the disciplines seems unjustified. Other issues which do not promote an independent software engineering discipline include the relative immaturity of the software engineering principles, the lack of appreciation among undergraduate students for the problems that software engineering is trying to solve, the general cognitive immaturity of the undergraduate students, the difficult interdisciplinary nature of software engineering, and the diverse application domain. There are currently no complete, separate undergraduate software engineering programs in the US (although at least two are on the drawing board).

However, there is one approach which has proven successful in producing a significant software engineering emphasis within an undergraduate computer science program. This approach is to migrate software engineering topics back from the masters curriculum into the senior year of an undergraduate computer science program, and eventually into the earlier stages of a computer science program. (Note that a phased "Program Evolution Strategy" for a top down evolution of software engineering within a computer science program has just been published by the SEI.) Many universities have taken this approach, but most have stopped after migrating just a course or two into the undergraduate program. Although a good start, the migration of optional courses does not meet the goal of producing professionals knowledgeable in software engineering. It is not sufficient to layer software engineering as an application on top of the computer science core; rather, a "software engineering core" must be developed and included along with the computer science core.

The future will see more of this migration into the lower division courses of computer science programs. However, for this method of developing an undergraduate software engineering program to succeed, the issues noted above must still be explicitly resolved. Additionally, from our experience, substantial emphasis beyond what is currently within a computer science program will be required for: issues of scale, analysis and design experience, Ada, and software development environments and methodologies. The new computer science/software engineering programs will also require a variety of courses tangential to the area of computer science but which play a role in software engineering, such as management, communications, and engineering design. They will also require a significant coordination of topics, concepts, and tools within the program and a renewed emphasis on faculty development and coordination. This evolutionary approach will continue to allow accreditation within the criteria of the Computer Science Accreditation Commission and will encourage the Commission to draft new criteria to encompass even more software engineering oriented programs in the future.

Position Paper: Curriculum Trends in Software Engineering

Gary Ford
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

The emergence of software engineering as a discipline mirrors the emergence of computer science and of the various engineering disciplines. In very broad terms, the process begins with increasing knowledge in a specialized area within an existing discipline, followed by the introduction of university courses at the graduate level, the emergence of separate master's programs, the organization and codification of the fundamentals and principles of the discipline, the infusion of the fundamental material into the undergraduate curriculum, and finally the emergence of separate bachelor's programs. Most universities are at the second phase of this process, but over the next decade, some will advance to the final phase.

Since its creation early in 1985, the Education Program of the Software Engineering Institute has worked to accelerate and assist the development of master's level software engineering education. This will continue throughout the 1990s, because almost all universities interested in this discipline will begin with graduate programs. SEI recommendations for these programs are contained in the annual *SEI Report on Graduate Software Engineering Education.*

Within the next five years, we expect to see the first undergraduate software engineering programs established. These programs will develop slowly, partly because of the lack of good textbooks and faculty expertise, and partly because much of the knowledge within the discipline is still being developed. Discussions of the design and evolution of undergraduate programs are presented in the annual *SEI Report on Undergraduate Software Engineering Education.*

Perhaps the most critical factor in the development of undergraduate programs will be the maturation and acceptance of software engineering as a true engineering discipline. Programming, as currently taught in almost all computer science programs, is a craft not at all closely related to engineering. In the traditional disciplines, engineers employ a number of mathematical and analytical techniques that allow designs with predictable strength, behavior, performance, and reliability. Software engineering is just beginning to develop similar capabilities.

A second critical factor is the necessary growth of faculty expertise in software engineering. Although computer science research 30 years ago was firmly grounded in real-world problems, current university research is increasingly aimed at solving abstract problems invented by *scientists* who are not well connected to today's software *engineering* problems. Faculty will have to adopt the philosophy and methods of engineering, both to teach and to conduct appropriate engineering research, if undergraduate programs are to emerge.

Accreditation of undergraduate programs in software engineering is at least five and probably ten years away. The accrediting bodies in computing, the Accreditation Board for Engineering and Technology and the Computing Science Accreditation Board, only accredit programs that have already produced graduates. Furthermore, they rely on accreditation criteria drafted by appropriate professional societies. Recent ACM and IEEE Computer Society curriculum efforts indicate that the societies do not yet recognize software engineering as an emerging discipline.

Nearly all of the significant issues in software engineering are unrelated to coding. However, neither computer science nor programming will cease to exist when software engineering has matured as a discipline and a profession, nor will software engineers be able to perform well without a reasonable knowledge of programming. Ada embodies most of the good programming ideas of the 1970s, and is the most widely available language that does. It is appropriate for programmers, computer scientists, and software engineers to have some familiarity with it.

This work was sponsored by the U.S. Department of Defense.

Position Paper: The Role of Ada in the Curriculum
Fifth ASEET Symposium,
Alexandria, VA, August 1990

Michael B. Feldman
Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, DC 20052
202-994-5253
mfeldman@seas.gwu.edu

Ada's role in GW's curriculum goes back to 1982. Ada has been the required coding language in our CS2-style course since 1985. At the graduate level, Ada is a (large) part of our required programming languages course, and comes into play in software engineering, concurrency, and compiler construction courses as well.

Most readers of this position paper will know that I am something of a "fan" of Ada. Surely Ada has many advantages to offer. It's not hard to teach to first-year students, given the right books—which are finally starting to emerge—and a sympathetic approach to schools on the part of the Ada compiler vendors—which is still very uneven but a lot better than it used to be. And Ada is a language that students can "grow into," which means it *can* be used as the language of most of the courses further up in the curriculum.

Perhaps the most significant contribution of Ada to education is that the standard is meaningfully enforced. Ada compilers support a language which is *really* common across different machines and vendors. This is a breath of fresh air to anyone who has tried to get upperclassmen, spoon-fed on Mr. Kahn's Turbo whatzit, to move to Unix Pascal, or to port a non-trivial program from one Modula-2 compiler to another. With Ada, the same books, the same knowledge, the same teacher experience, and with some care, the same programs, will work on *any* computer in the place (well, almost: we're still all waiting for that cheap and fast Macintosh Ada compiler...). This commonality—acceptance of a real standard, détente in the "feature wars"—is more than a convenience: it is a important lesson for our students in maturation and professionalization of the discipline.

All this notwithstanding, I think we need to keep our perspective. What should be the role of *any* programming language in the curriculum? The students tend to place *great* emphasis on the language used in a given course. Every student, seemingly, needs to know, in advance, "which language we'll use." The students may be misplacing their emphasis; the teachers don't have to go along with it. True, one usually needs a definite, compilable, language of discourse for a class. But it shouldn't be the primary consideration.

Academic computer science has for *decades* been wracked by dissension over the "breadth vs. depth in languages" issue. I also hear mixed messages about this from my industry friends. My position? We are educating professionals, not training worker bees, and therefore we need to understand some truths about our chosen field:

- For forty years, one of the few constants in the computing field has been *rampant change*. This is true of hardware, software, languages, and everything else, and no end is in sight.

- Professionals need to have a solid grounding in the basic principles of computing, including theory, architecture, and software, but also need—above all, perhaps—to have a mind which is ready for change and open to change.

- Coding is only a part of software engineering: language power surely makes coding easier, but we cannot fall into the trap the students set for us by emphasizing the details of today's annointed language over the larger issues.

- Ada is—should be—today's language of choice for many in industry, but there are other strong contenders out there too. With luck, the proponents of emerging languages will *follow* Ada's lead and adopt meaningful standards. But in any case, whether DoD likes it or not, in ten years I'm sure we shall look back on Ada as a rather primitive creature. I believe that DoD needs stability, not change, and so needs to stick with Ada, but the computing world's bigger than DoD.

At GW we have adopted and sustained, over many years, a "breadth" preference. Our undergraduates learn—in the "natural" courses—Pascal, Mac Assembler, Ada, C, Fortran, and Lisp. They may not be real experts in any of these, but that kind of expertise comes in the workplace anyway. If we are doing our job right, their *real* expertise is in seeing that languages are all different but in a way all alike, that the coding language plays an important but not exclusive role in software engineering.

The computing world is best served by college graduates—and college teachers—who are multilingual and ready for change, not by "Pascal bigots" or "C bigots" or "Lisp bigots"—or, indeed, "Ada bigots." Our students need a sense of proportion and balance. If they can't measure up to this standard, they'd be better off in another field. Teachers—we, here, now—need proportion and balance too.

The Place of Ada in an Undergraduate Software Engineering Curriculum

Luwana S. Clever Charles B. Engle, Jr.

20 June 1990

Introduction

The Florida Institute of Technology (FIT) will introduce a new undergraduate curriculum in software engineering in the fall of 1990[1]. Much time and effort has been put into the selection of the "right" topics to cover and the order in which to present them. It is now time to discuss some of the engineering trade-offs that were made and why.

A complete description of the FIT undergraduate curriculum can be found in [Mills90a]. This paper, which was presented at the Software Engineering Institute's Conference on Software Engineering Education, discusses the role of universities in software engineering education. Detailed

[1]The opinions expressed here are those of the authors and may not accurately reflect those of all faculty members concerned with this project. Care has been taken to attempt to represent the consensus of opinion, but since the program will not start until Fall '90, anything mentioned in this paper is still tentative

listings of courses and course content can be found in this paper. In addition, considerations of transitional courses for current computer science majors and "service courses" for other departments are discussed.

The methodology that will be used at FIT for undergraduate instruction is the *cleanroom* software engineering techniques pioneered by Dr. Harlan Mills [Mills90b], and box structures [Mills87, Mills88]. Complete discussion of this methodology, and its underlying philosophy, is beyond the scope of this paper. Instead, this paper is limited to a discussion of why the decision was made to use Ada as the language of choice for *all* undergraduate software engineering instruction. This was not an easy decision. There are many that would claim that Ada is "too complex" or perhaps "too big" to be used for freshman level instruction. We believe that this is a false impression.

Note that we do *NOT* advocate the use of a *Pascal subset* of Ada. Rather, we believe that students should be exposed to concepts and ideas which support good design and software development practices using a software engineering approach. The student should be given the tools with which to implement these concepts at the time that the concept is discussed. This means that students quickly outgrow Pascal and need the more powerful features and constructs in Ada to support the concepts that they are learning in class.

For example, in programs which use interactive input-output, there is

little protection for the Pascal programmer if data entry is erroneous. The program simply *bombs*! The alternative is to teach the student to read everything as a character string and then do the conversions to the appropriate type in the program. This is awkward and unnecessary since it is not a part of the solution to the original problem, but an artifice of the limitations of the programming language.

With Ada, of course, the proper means for solving this problem would be to design the program as a solution to the original problem without the need to compensate directly for potential input-output errors. Incorrect data entry would automatically be detected by the runtime system and can then be handled appropriately by the student within the bounds of the programming language without the unnecessary, even unnatural, contortions required by Pascal.

We studied this and other problems in teaching students a "smaller, simpler" language first, followed by a transition to a more powerful language when necessary. We concluded that the attendant mind-shifts were more difficult for the average student than providing him or her with the proper tool from the very beginning. The factors which weighed in this decision are described in the remainder of this paper.

Advantages of Ada

Ada offers many advantages as a base language for our curriculum. It is rich enough to be useful for most programming concepts. It is a practical language, finding widespread acceptance in government, and thus, industry. It is an expressive language allowing less constrained design transformations. It is widely available as a *standard* which is certified correct (actually certified as conforming). It was designed, even engineered, to support the very concepts of software engineering that we are teaching our students. For these, and many other reasons, Ada has been adopted as the base language for our curriculum.

Ada is the first programming language that was created with the *engineering process*. It was *engineered* to support good software engineering practices and concepts as we understand them. As such, Ada was *not* an evolutionary language wherein new features are spliced onto a baseline shell. Rather, the Ada language was designed and developed like any other software product. This effort was initiated by soliciting and obtaining a series of requirements for the new language. These requirements were then refined by widespread public review into a set of specifications upon which a design could be developed. This preliminary design was also given widespread review and a final design for the language was approved, *before any implementations of the language existed.* This was a novel concept in

the design of programming languages; obtain consensus on the requirements before implementing it! This shows that Ada was designed and engineered to perform specific functions, prominent among them was the support of software engineering concepts.

The concepts which Ada was expressly designed to support, according to [Booch87], include abstraction, information hiding, localization, completeness, modularity, reliability, maintainability, reusability, and extendability, among others. This list gives credence to the claim that the design of Ada was intended to support modern software engineering concepts and practices as we understand them. Arguably, the implementation of the language, manifest in numerous compilers on numerous machine configurations, provides the much needed support for Software Engineering that has been missing in older languages.

The support of modern software engineering practices and concepts is very important. If a language is very rich in expressivity, then it becomes less difficult and less error prone to translate the problem to be solved from the design space to the solution space. If the language is somewhat limited or constrained in its expressive power, then the mapping from the design space to solution space is more difficult. For example, if the design of a solution to the problem at hand conceptually requires the abstract notion of parallelism, then if the language in which the design is being implemented supports parallelism, this portion of the solution can be directly mapped

from the design to the implementation. If, on the other hand, your language does *NOT* support parallelism, then you must serialize your conceptual parallelism, which means that you must introduce additional complexity into the implementation to achieve the effect of your design. This necessarily perturbs the design and makes maintenance more difficult. In summary, the more powerful the language in terms of expressivity, the more easily you can map the design to the implementation without introducing additional complexity [Mills90a].

In view of the foregoing, the rich set of constructs and programming expressivity available in Ada make it the logical choice for our curriculum. While some may argue that the language is "too big" or "too complex" for freshman, we take the opposite view. It is our view that we need only acquaint the student with that portion of the language which is necessary for them to solve the problems that we provide. In time, this will be the full language. What we obtain from this is the ability to go from simple sequential concepts to more complex ideas, such as parallelism or genericity, without having to transition the student from a smaller, less powerful language to Ada. They will have been using the same language since the first programming assignment!

Disadvantages of Ada

It would be unfair, or at least one-sided, to present only the positive aspects of the use of Ada. We acknowledge some undesirable items that we must either overcome or for which allowances must be made. We will mention some of the more prominent disadvantages that we have encountered and what, if anything, we have decided to do to compensate for these shortcomings.

Perhaps the biggest disadvantage of using Ada, at least from our perspective, is that there is a rather significant overhead penalty which must be paid, even for small, simple programs. This overhead is due to the loading of runtime routines providing exception detection, and possibly handling, as well as the loading of the support system for parallel program units (tasking), even if they are not used in the unit being compiled. In fact, a complete runtime system capable of operating *sans* operating system is provided by the Ada compiler. Thus, a small program may well cause the generation of a substantial executable image!

In large programming systems, this overhead is usually insignificant. However, at the freshman level, where programs are very short and very simple, this overhead is unusually expensive relative to the size of the program. Not only will the executable image be disproportionally large, but the time to compile the code will be longer than expected due to the added

time needed to generate the load module. Usually even the execution time will be adversely affected by this unnecessary overhead.

Since this runtime overhead is a requirement of the Ada language, there is little that we can do to avoid it. In the absence of some compiler directive or *pragma* which allows the suppression of the linking of these unnecessary runtime modules, we can only mention them to the student. As time goes on, the proportion of this runtime overhead to their program will decrease, until, eventually, when they are writing programming systems instead of programs, they will not even notice the overhead. Until then, there is nothing that we can do.

Another shortcoming of Ada is the fact that input-output (I/O) is not a part of the language. Instead, the I/O features were supplied by a predefined package. This makes good sense since the package is the natural way to extend the language . However, it does cause some difficulties as any instructor that has taught Ada is well aware.

First, the absence of I/O for numeric types necessitates that the user create a generic instantiation of a predefined numeric I/O package in order to be able to read or write numeric values. The particular reasons for this mechanism of I/O are not within the scope of this paper, except to say that it is a prudent and appropriate means of providing a robust, even efficient, I/O mechanism. However, the instructor is typically left with a chicken-and-egg situation with regard to numeric I/O. In order to do I/O

the student must be familiar with generics (or at least generic instantiation), but the need for numeric I/O occurs long before the introduction of the topic of generics. The problem heretofore has been how to overcome this difficulty.

Numerous solutions have been presented. They range from the bad to the worse. One method is to launch into a description of generics early, with some limited specification of what they are and how they are used, followed by a more complete discussion at a later time. This can be confusing to the student, but if handled properly, can be compared to Boehm's spiral method [Boehm88] for software development, and made quite effective.

Another method is to show the student the format for generic instantiation with little or no explanation. This saves confusion. On the other hand, the *trust me, I'm the instructor* method is not pedagogically satisfying. It makes the whole process appear to be done with smoke and mirrors, as if by magic!

Another approach is to define a package for the students with all of the predefined numeric types *pre-instantiated.* This allows the instructor to refer to a Basic_IO package and not have to explain numeric I/O instantiations. The student merely uses the Basic_IO package and all of the I/O they need is provided. The drawback to this method is, of course, that *only* the predefined numeric types can be handled in this manner. This, then, restricts the instructor to using only the predefined numeric types, and lim-

its the use of good software engineering practices wherein the student will specify new types for each new situation and perform a domain analysis to determine ranges of values and degrees of precision. Thus, pedagogically, this is not a viable solution in our environment. We reasoned that there must be other ways.

Our solution to this problem is a compromise with our desires and the design of Ada. We will teach the students to use *subtypes* of the predefined numeric types, wherein the subtype will have a unique name and a range or precision as determined by domain analysis. However, since they have defined a subtype and not a type, compatibility remains for all existing predefined operations. The use of a Basic_IO package is now a possibility owing to the compatibility of the subtype for the base type. This method does not allow Ada to enforce the prohibition of the mixing of dissimilar types, since all of the types are subtypes, but it is an acceptable compromise with which to start. Later, we will introduce the use of types and at that time introduce the concept of generic instantiation for these types.

Finally, another shortcoming of Ada that has slowed its introduction into mainstream computer science/software engineering programs is the lack of a good textbook at the CS1 level [ACM78]. This we see as a short term problem since new textbooks are being introduced at an increasing rate. Several textbooks on the horizon have made a commitment to using Ada in CS1 level instruction. Another more immediately available alter-

native is to write your own textbook. That is the solution that we chose, since it allows us to introduce Ada, but more importantly, it allows us to introduce software engineering in a way that we think is appropriate for our students.

I Can't Use Ada Because ...

One of the arguments made against the use of Ada on such a large scale at the undergraduate level is the lack of availability of compilers. To counter this problem, we contacted several compiler vendors. We found that there were actually several compiler vendors that were willing to make very attractive arrangements with an educational institution. After careful consideration we chose to accept the offer of Meridian Software Systems, Inc.

In essence, the arrangement with Meridian allows our students to purchase fully validated Ada compilers for their IBM-compatible machines. The compilers can be obtained on $3\frac{1}{2}$" or $5\frac{1}{4}$" diskettes and are provided with sufficient documentation for easy installation and use. The compilers, though fully validated, are student editions. This merely means that the number of libraries is limited (to two) and that the number of units that can be placed in the library is limited (to approximately 200). Neither of these limitations is likely to affect student programs; indeed, we use these same

compilers for our graduate students in their coursework without significant limitation. The students are required to pay $50.00 for this compiler.

Meridian has recently notified us that they will also initiate this summer an Ada student version for Macintosh computers. This version of their compiler will also cost $50.00, but will require that the student already own Macintosh Programmer's Workbench (MPW). This is necessary because Meridian's Macintosh version actually runs under MPW. In any case, this action by Meridian will extend the accessibility of Ada compilers for our students.

The process used to obtain the compilers is simplicity itself. On the first day of class we describe the compiler to the students and recommend that they purchase it. A count is taken of those students that want to obtain a compiler. The instructor then calls Meridian (using their toll-free number) and tells the educational assistant how many compilers are needed. The compilers are then shipped via overnight mail and arrive in time for the student's next class. Each student then pays for the compiler as they pick it up.

The logistics of this mode of operation are attractive to Meridian because they can sell multiple compilers with only one shipping and handling operation. For this reason, they sell the compiler *to the school* for $40.00. Thus, even when the school pays for the shipping charge (a flat $25.00), there is still some excess money which can be used to purchase additional

compilers for lab machines, etc. This arrangement is a mutually beneficial deal for students, the school, and Meridian.

We fully realize that this sort of arrangement is possible because we are a private institution. Public institutions may not be able to avail themselves of this opportunity, due primarily to legal considerations. In that case, each student can deal directly with Meridian for the purchase of their compiler.

Note also that we *never* force a student to purchase this, or any other, compiler. The option is always available for the student to purchase a different compiler or to use one of the compilers provided on the lab machines.

Another argument made against the use of Ada is that there are insufficient environment tools available for student use. We also found, and are currently negotiating for the use of, a complete environment tool which features many useful support aids for our students. Among these are an on-line version of the Reference Manual, an on-line help facility, interactive use of the compiler, an emacs-like editor, CAI, randomly generated examination questions with automated grading and record keeping, password protection on all directories, and the use of a network and communication facilities so that a single copy of the environment is needed. We have not yet made a firm commitment for the use of this or any other environment so we cannot offer any more details at this time.

In short, we are introducing a new undergraduate curriculum and need new, more powerful tools to support it. We feel that the use of Ada as our

language of choice, supported by compilers and an environment, will augment our curriculum and make a more meaningful educational experience for our students.

Conclusion

We have tried to avoid the pitfall of many papers in this genre which read like a *this is what our program looks like.* Instead, we have attempted to explain *why* our curriculum designers made the decision to use Ada as our base programming language in our new undergraduate degree in software engineering. It is still our opinion that the firm commitment to Ada that we have made is the proper decision given all of the data we assembled. Time alone will tell if we are correct!

Bibliography

ACM78 Austing, R.H., Barnes, B.H., Bonnette, D.T., Engel, G.L., and Stokes, G. "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science." *Comm. ACM 22*,3 (Mar. 1979), 147-165.

Boehm88 Boehm, B.W., and Papaccio, P.N. "Understanding and Controlling Software Costs." *Transactions on Software Engineering 14*,10 (Oct. 1988), 1462-1477.

Booch87 Booch, G. *Software Engineering with Ada.* Menlo Park: Benjamin/Cummings, 1987.

Mills87 Mills, H.D., Linger, R.C., and Hevner, A.R. "Box Structured Information Systems." *IBM Systems Journal 26*,4 (Apr. 1987), 395-413.

Mills88 Mills, H.D. "Stepwise Refinement and Verification in Box-Structured Systems." *Computer 21*,6 (Jun. 1988), 23-36.

Mills90a Mills, H.D., Newman, J.R., and Engle, C.B. Jr. "An Undergraduate Curriculum in Software Engineering." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1990, 24-37.

Mills90b Mills, H.D. "Cleanroom Software Engineering." *Aerospace Software Engineering*, C. Anderson, M. Dorfman, and D. Hall, eds. AIAA, 1990.

This Page Left Blank Intentionally

PROGRAM DEVELOPMENT AND ADA

A Paradigm for Teaching Programming to Lower Level Computer Science Majors

Robert A. Willis Jr.
Hampton University
Department Of Computer Science
Hampton, VA 23668

Introduction

Despite the best efforts of their instructors and the formidable motivation found in most modern texts, freshman and sophomore computer science students tend not to follow any program development methodology when writing programs.

The first few programming assignments are usually simple and small. They are attuned to building confidence and developing manual skills. Students develop a false sense of security, they "know" that they can successfully write programs while ignoring the foundations of good programming practices. Their delivered products are properly formatted and commented, the programs "look good". But what about the development process? Did the students do any significant "front-end" work? All too frequently the answer is "NO" and as their program assignments become linearly more difficult, students find that the intellectual effort required to complete assignments increases exponentially.

Asking students to deliver specifications and a top-down design is not enough. These documents are often written after the program is finished, and if we (the instructors) request them prior to the coding stage many students have more difficulty creating a complete design than the actual program itself! The concepts of step-wise refinement, modularity and top-down design have to be presented to students in a manner which fosters understanding and appreciation of their elegance and power. This paper discusses a paradigm that seeks to inculcate the use of proper program development in our students.

The Concept of Incremental Program Development (IPD)

When we assign a programming exercise to our students, we generally include a comprehensive specification, one or more data files (or a sample of expected output), and a due date. We then explain the exercise and perhaps "walk through" the basic processing steps. It is now up to the students to write, test, and deliver the program by the due date. We are all aware of the agonizing process that most of our students go through. Incremental program development assures the use of step-wise refinement, modularity, and top-down development by our students.

Instead of one due date students are assigned a series of intermediate due dates, each with specific expected results. Each ensuing delivery encompasses progressively more difficult aspects of the assignment. Students are expected to present a design and a working program at each stage. This incremental delivery process forces students to focus on one specific aspect of the problem at a time (step-wise refinement) and to mentally and physically (design and code) solve the problem beginning with a fundamental system model (top-down development). If we are careful in the stipulation of expected results for intermediate deliveries our students receive invaluable "hands on" training regarding step-wise refinement, modularity, and top-down development. Equally important is the immediate feedback the student receives with each delivery. Critical design errors and instances of non-conformance to specifications are found early and can be easily corrected.

When the final due date arrives the entire program is delivered and a grade is given.

Why Ada?

I have used IPD in courses featuring both Pascal and Ada and found that Ada is the superior tool when using the IPD paradigm.

FIGURE 1. A First Programming Example

```
-- in the following section we obtain the modules we need to write text,
-- and real (numbers) to the screen. Remember that these entities are
-- not the same and require different procedures (modules) to write them to
-- your screen. We may use the same names but the actual code is different,
-- since these modules are given to us we don't see the code, we just use
-- the modules.

with Text_IO; -- Write text to the screen
with Float_Text_IO;-- Write real (float in Ada) numbers to the screen

procedure One is
  --
  -- Programming Assignment One
  -- Your name
  -- date copied and run
  -- description of problem

  Debt: constant:= 300.0;-- Original value owed
  Payment: constant:= 22.4;-- Payment
  InterestRate: constant:= 0.02;-- Interest rate

  Charge,-- Interest times debt
  Reduced,-- Amount debt is reduced
  RemainingBalance: Float;-- Remaining balance

begin -- One
  Charge:= InterestRate * Debt;
  Reduced:= Payment - Charge;
  RemainingBalance:= Debt - Reduced;
  Text_IO.Put ("Payment ");-- Note that we use the Text_IO module here.
  Float_Text_IO.Put (Payment); -- Note that we use the Float_Text_IO module.
  Text_IO.New_Line;
  Text_IO.Put ("Charge ");
  Float_Text_IO.Put (Charge);
  Text_IO.New_Line;
  Text_IO.Put ("Balance Owned ");
  Float_Text_IO.Put (RemainingBalance);
end One;
```

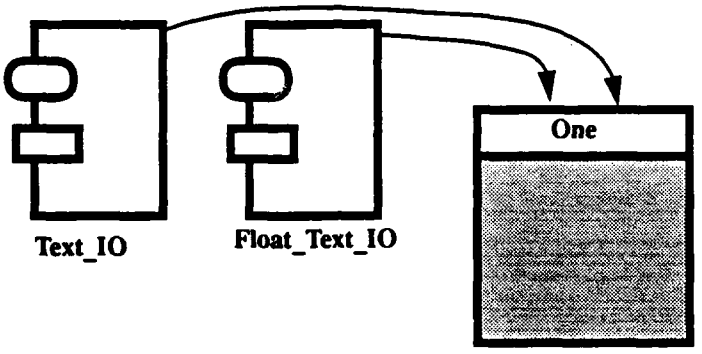
Modularity

Students initially benefit most from Ada's concept of modularity. Simply writing the classic "Hello World" program is a good illustration. The program must import Text_IO in order to produce output. As instructors, we can use this necessity as a small case history concerning modularity, explain what modularity is, why the benefits far outweigh the trouble of using with statements, and package instantiations (if necessary). We needn't discuss what a package instantiation is, or go into the gory details of what Text_IO really is, just enough to show that modularity is an important concep-

tual tool. Students are forced to think in terms of single purpose units.

The program[1] shown in figure 1 is an exercise which explicitly illustrates the point. Students are required to input and run it. The example purposely uses the module **Float_Text_IO**[2] rather than instantiate a new package in the body of the program unit. I have found that this method is easier understood yet still conveys the essence of modularity. Here the essential issue is that text and real numbers are different, that it requires different code to write them to a terminal screen, therefore we need separate modules to perform the work. Using the qualifiers **Text_IO** and **Float_Text_IO** further demonstrates that different modules are in use. Figure 2 depicts a possible top level graphical design.

FIGURE 2. Top Level Graphical Design for Program Unit: One



Ada allows us to implant the idea and convey the significance of modularity to students with its very first examples of working programs. It is important to note that this is done naturally and in a form which (since it is part of the language) is entirely consistent with later use.

Of course what we are really exploiting are Ada's separate compilation and data abstraction (packages) features. But we do it in a manner which is relatively "painless" to students. We are also beginning to build towards a more complete understanding of data abstraction than is usually possible when using Pascal.

It is very difficult to properly treat data abstraction with Pascal. Procedures in Pascal are more properly process abstractions than data abstractions and most students have difficulty discriminating between the two. Process abstraction is rarely discussed, but discussions of data abstraction and the examples usually given (input/output and some standard mathematical functions) while excellent examples of data abstraction consistently miss the mark because their use is so transparent. Students are simply not yet sophisticated enough to fully grasp the concept and when confronted

1. Dale, Nell and Weems, Chip, "Introduction to Pascal and Structured Design", P.92 2nd Edition D.C. Heath and Company (1987)
2. Digital Equipment Corporation Ada for VAX/VMS

with Pascal's lack of true abstraction many are more confused about data abstraction than enlightened.

In CS 2 the lack of true data abstraction in Pascal is particularly bothersome. One encounters problems whenever data structures are discussed. The **stack** as an abstract data type is an excellent example of this. The concept of a stack can be completely specified and thoroughly explained to each students satisfaction. The Pascal implementation, however, **undermines** the conceptual model simply because the data type *StackType* and all stack operations must be declared as global entities in the using program unit. Students often question this discrepancy and regardless of the explanation relegate data abstraction to the "ivory tower concept often preached but never practiced" category. Worse yet, many of them attempt to take advantage of their knowledge of the implementation and write code which manipulates the stack from their program, instead of using the proper procedures.

A preliminary survey[3] of CS 2 students revealed that 55% accessed the stack (or other abstract date type) using its operational procedures **only** because the instructor indicated that it was necessary. Only 20% of the students could fully explain the concept of abstract data types. The same survey given to a group of students in a beginning Ada class revealed that 85% could fully explain the concept and 95% appreciated the benefits of its use. It should be noted that 97% of the Ada students had completed CS 2, but a sample survey of CS 7 (Data Structures and Algorithms) students who **did not** take Ada indicates that the majority of them had significant problems with the concept of data abstraction.

Exception Handling

Ada's exception handling facilities are invaluable as a program development tool. Unlike Pascal, which does not implement exception handling, they allow the instructor to discuss program robustness and graceful termination in a rational manner. It is very difficult to illustrate program robustness in Pascal because examples typically demonstrate specific solutions to a particular problem rather than depict general solutions to a class of problems. Worse yet, the programmer is often forced to obfuscate his code significantly in order to create a robust product.

Ada presents beginning students with a "clean" uncomplicated set of predefined exceptions, each of which addresses a class of errors which commonly occur. It also allows more advanced students to define their own classes of exceptions. Students may elegantly include error handling as an integral part of their designs.

IPD and Ada

Critical to any program development is the design phase. In this phase, program specifications and

3. A small sample survey. A larger sampling of students from at least two different Computer Science Departments will be conducted shortly.

delivery schedule show students how a problem can be partitioned and solved, in other words, how to approach the design of a program. An extremely important facet of the IPD paradigm, is that students are continually receiving concrete guidance in program design.

Equally important the specifications usually provide a good indication as to how the program should be modularized, when an Abstract Data Type (ADT) should be defined, and the type of exception handling which will be required. All of these can be directly translated into Ada code. Modules can be built, separately compiled and tested as necessary. ADT's can be similarly designed, implemented, and tested without regard to the logic of any specific application. User defined types can be properly defined and I/O instantiated for them. This allows students to build-in true exception handling and begin to learn what reliability is all about.

An Example of Incremental Program Development[4]

1.0 Design and write a program to solve the following problem:

1.1 Eastern Shore Airlines schedules 6 daily flights from Salisbury, Maryland to Dover, Delaware. Demand for these flights is so high that customers are placed in one waiting list and will only obtain a flight in turn. Create and maintain a flight list for the airline. Each flight carries a maximum of 5 persons.

2.0 Input:

2.1 Filename: Flight_Data.DAT.

2.2 Each line of input contains:

- the request: "DEPART", "ADD", "CANCEL", "FIND", or "LIST" in columns 1 -6
- and unless "LIST" or "DEPART" is requested, the passenger's name in columns 12 - 31.
- You may assume that the format is correct.

3.0 Processing:

3.1 The five types of requests should be handled as follows:

- If the request is "DEPART," then the next flight will depart. The flight number and a list of passengers is also printed.
- If the request is "ADD," then the given passenger wants to be added to the flight list.

---

4. Collins, William J., "Intermediate Pascal Programming: A Case Study Approach", McGraw-Hill (1986)

- If the request is "CANCEL," then the given passenger should be removed from the flight list.
- If the request is "FIND," the value output should be "YES - ON FLIGHT LIST" or "NOT ON FLIGHT LIST."
- If the request is "LIST," then the flight list should be printed.

3.2 The following exceptional cases should be noted:

- If the request is not one of the five described above, the error message "ERROR-ILLEGAL REQUEST" should be printed.
- If the request is "DEPART" and the passenger list is empty, then the message "EMPTY FLIGHT flight_number DEPARTING" should be printed.
- If the request is "ADD" but the passenger is already on the flight list, then the error message "ERROR - DUPLICATE NAME" should be printed.
- If the request is "CANCEL" but the passenger is not on the flight list, then the error message "ERROR - NAME NOT ON FLIGHT LIST" should be printed.

3.3 Processing ends when the last flight has departed.

4.0 Output:

4.1 All legal messages and conditions for output are detailed in the previous section. Please ensure that your output is neat and readable.

5.0 Delivery dates:(21 days total elapsed time)

5.1 First delivery date (3 days elapsed time)

- The program should be able to read and echo print each command found in the data file.
- Simulate processing by issuing appropriate statements for each command.
- Implement exception handling for all illegal command.

5.2 Second delivery date (2 days elapsed time)

- The program should now be able to print all passenger names. A passenger's name is printed to the immediate right of the appropriate command.

5.3 Third delivery date (5 days elapsed time)

- The "ADD" and "LIST" commands are implemented. Duplicate names are allowed.

5.4 Fourth delivery date (5 days elapsed time)

- The "DEPART" and "FIND" commands are implemented. Duplicate names are no longer allowed.

5.5 Final delivery (6 days elapsed time)

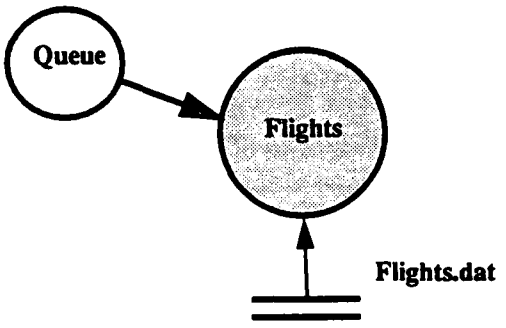
- The "CANCEL" command is implemented.

5.6 An appropriate design will accompany each delivery.

The First Delivery

The first delivery forces students to consider the basic logical structure of the finished product without concern for lower level details. A possible graphical design for the initial delivery is depicted in the following figures. All students are expected to present a set of designs with their initial delivery. Arriving at this set of designs should be easy for them because the delivery does not require detailed lower level processing. They are focussing only on basic input and error handling.

FIGURE 3. Top Level Design: Flights



Given the description of the problem (and recent class discussions) it should be clear that a queue data structure will be necessary. The top level design indicates this and the fact that it will be separate entity (what type of entity is not decided here). Top level designs not including it are considered deficient.

FIGURE 4. Second Level Refinement: Flights

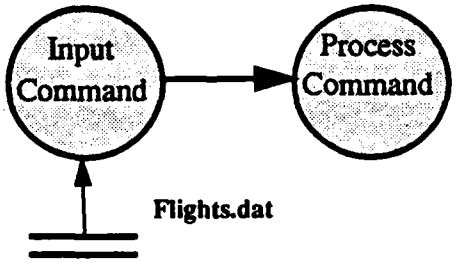
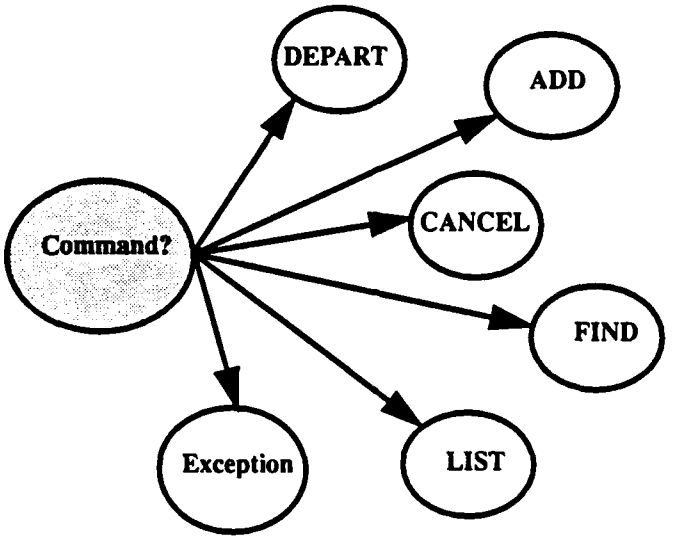


FIGURE 5. Further Refinement: Process Command



We have now arrived at a design which fully complies with its specification. Fortunately, the delivery specification was posed so that the problem of designing the basic processing of the program was forced upon students at an early stage. The design details for this delivery are really the high-level design details for the final product. This clearly illustrates the power of IPD to motivate top-down design. These points need to be emphasized to the students and an example of a bad delivery specification and ensuing problems should be examined.

The design can now be translated to Ada code and tested thoroughly. If testing reveals any flaws the design can be revised. Revising the design (and subsequent code) at this stage is relatively inexpensive because dependent lower level details have been excluded from the design and code

Going to code at this early a stage in program development is also important in that it provides students a controlled method to satisfy their urge to start generating code. Code generation in this case is not premature because it logically follows the development of a valid design. Students are more apt to actually design first because they are not designing the entire program at once, just a small manageable portion (step-wise refinement).

The following figure contains code based on the design. No references are made to a queue to simplify delivery. Please note that the more ambitious students may be inclined to include procedure stubs and actual calls to them from within the case statement. This is erroneous, in that a raft of antecessor (at this level) details must be considered. Issues such as interfacing, file manipulation, among others should not be considered, at this time, because they hide a host of subordinate details. Remember that, at this stage, we are interested in a high level design and in general, we are attempting to develop the program incrementally.

FIGURE 6. Flights.ada

```
with Text_IO;
use Text_IO;
procedure Flights is
--
-- **** This delivery of the program should be able to read and echo print
-- each command found in the data file.
--Simulate processing by issuing appropriate statements for each
--command.Implement exception handling for all illegal commands.
--
--    Which means that the basic intelligence of the program must be
--    implemented. This is accomplished by the use of a case statement

  type Command_Type is (DEPART, ADD, CANCEL, FIND, LIST); -- List of commands
--
-- Packages required to manipulate Queues and Enumerated IO.
--
  package Command_Type_IO is
       new Enumeration_IO (Command_Type);
  use Command_Type_IO;
--
-- Variables
--
  Command: Command_Type;
  Input_File: File_Type;

begin -- Flights
  Open (Input_File, In_File, "Flight_Data.dat");
  --
  -- Processing begins. Use of case statements allows easy handling of
  -- the different commands.
  --
  while not End_Of_File (Input_File) loop
    begin -- Exception block for incorrect commands
      get (Input_File, Command);
      Put (Command);
      case Command is
        when DEPART => Put_Line ("Flight Departing");
        when ADD    => Put_Line ("Adding Passenger");
        when CANCEL => Put_Line ("Cancelling Passenger");
        when FIND   => Put_Line ("Finding Passenger");
        when LIST => Put_Line ("Flight List");
      end case;
      New_Line;
    exception
      when Data_Error =>
        Put ("ERRONE0US COMMAND");
        New_Line;
    end; -- Exception block for incorrect commands
    Skip_line (Input_File);
  end loop; -- while not End_Of_File
end Flights;
```

Remaining Deliveries

Remaining deliveries are focussed on implementing individual processes and the queue. Implementation of the queue can be delayed until delivery three. Ada provides a number of implementation options here. The processes can be written as in-line subprograms within Flights.ada as in Pascal. They can be written, separately compiled and linked to Flights as subunits or using the with statement. The queue will almost certainly be implemented as an abstract data type (Package), but students must decide whether to make it generic or not. Each of these approaches should be thoroughly discussed.

It is during this phase of the program's development that Ada is most beneficial. Using separate compilation (in this case subunits) students concentrate on the development of one module at a time. Each module will be designed, coded, tested, and finally included into the main program. Decisions concerning interfacing, internal data structures, and processing are made largely in isolation. If, during testing, faults are discovered the source can only be in the module being developed. Students enrolled in the lower level CS courses have a tendency to make wholesale changes to their code in order to "get something working". Too often when they correct one problem they create another.

Using Ada they recognize that a previously designed and tested increment is not the cause of the error because additions to the program are created in a totally separate module, therefore it makes no sense to attempt to correct it by modifying previously generated code. Instead, they should concentrate their efforts on the module being developed. Modularization in the form of separate compilation reduces the frequently overwhelming bulk of code confronting students and serves as a conduit to properly channel their efforts. Ada continually serves as a practical tool to illustrate many of the concepts we attempt to convey at the lower level. Students are not only being educated about these concepts but they are being correctly trained in their actualization. They are properly developing program construction and organizational skills using a modern, no compromise, "real-world" mechanism.

Delivery Three

Implementing the Queue

Space does not permit complete development of the Flights program, therefore only the development of increment three will be illustrated. This delivery mandates the implementation of a queue, the provision to **ADD** a passenger's name into the queue, and the ability to **LIST** its contents.

If a generic queue package has been developed and tested, the decision should be to reuse it. I advise this strategy, because it serves as a good example of reuse, as well as, the elegance of abstraction and its usefulness when fully incorporated into a programming language.

The following figure contains the generic queue's specification.

FIGURE 7. Specification: Generic Queue

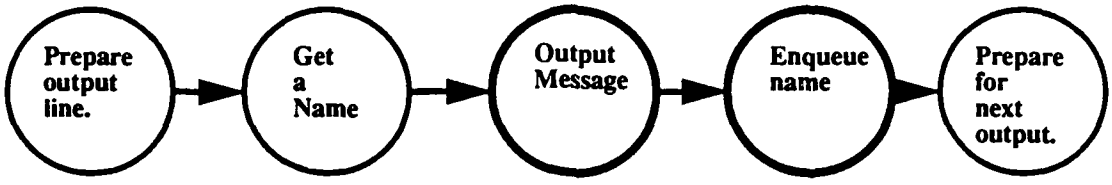
```
generic

  type ITEM_TYPE is private;

package GENERIC_QUEUE_PACKAGE is
--
-- This is a rudimentary generic queue package. The queue is implemented
-- as a linked list with the type of elements determined by the
-- user.
--
-- USER INSTANTIATION NOTES:
--
--package SOME_QUEUE is
--new GENERIC_QUEUE_PACKAGE
--  (ITEM_TYPE => SOME_TYPE);
--
  type QUEUE_TYPE is private;

  procedure ENQUEUE (ITEM: in ITEM_TYPE; QUEUE: in out QUEUE_TYPE);
    -- Places item into the rear of the queue.

  procedure DEQUEUE (ITEM: out ITEM_TYPE; QUEUE: in out QUEUE_TYPE);
    -- Removes item from the front of the queue

  procedure CLEAR_QUEUE (QUEUE: in out QUEUE_TYPE);
    -- Clears the queue

  function IS_QUEUE_EMPTY (QUEUE: QUEUE_TYPE) return BOOLEAN;
    -- Returns TRUE if the queue is empty FALSE otherwise

  EMPTY_QUEUE_ERROR: exception; -- Raised when attempting to dequeue an
                    -- empty queue.

  private
    type QUEUE_NODE_TYPE;
    type QUEUE_NODE_POINTER_TYPE is access QUEUE_NODE_TYPE;
    type QUEUE_NODE_TYPE is record
                      ELEMENT: ITEM_TYPE;
                      NEXT: QUEUE_NODE_POINTER_TYPE;
                    end record;
    type QUEUE_TYPE is record
                  FRONT: QUEUE_NODE_POINTER_TYPE;
                  REAR: QUEUE_NODE_POINTER_TYPE;
                end record;

end GENERIC_QUEUE_PACKAGE;
```

Design Issues

Given the specifications for add and list and the state of the program after the second delivery, completing the design for this delivery should be relatively straight forward.

Sample designs for **ADD** and **LIST** are shown in the following figures.

FIGURE 8. Design: Module ADD



This is a case in which each design module can be nearly translated one-to-one into Ada code. In fact, the actual implementation of **ADD** can be made in-line.

FIGURE 9. Design: Module LIST

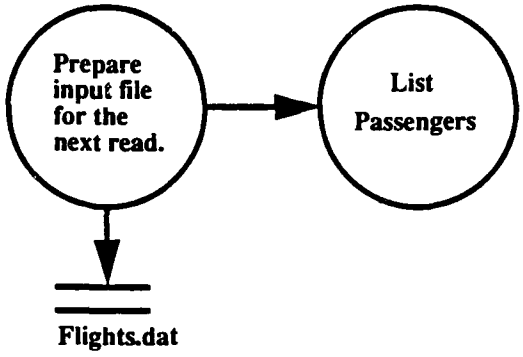
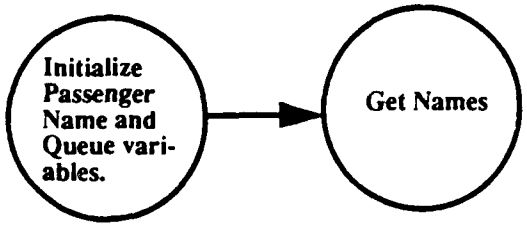


FIGURE 10. Design: SubModule: List Passengers



Get Names

Dequeue from a temporary Queue.

Print the name.

Quit when the temporary Queue is empty.

In this case it is advisable to create a procedure **List_Passengers** to output the names of the passengers.

FIGURE 11. Flights with ADD and LIST implemented.

```
with Generic_Queue_Package;
with Text_IO;
use Text_IO;
procedure Flights is
--
-- **** This is the third delivery ADD and LIST are implemented.
--
  Maximum_Name_Length: constant:= 20;
  --
  -- Column in input file at which the Name of a passenger starts.
  -- We have to use Text_IO.Positive_Count as the type because when
  -- we use the Set_Col procedure a number of this type
  -- is required.
  --
  Start_Of_Name: constant Text_IO.Positive_Count:= 12;

  subtype Name_Type is String (1..Maximum_Name_Length);
  Blank_Name: constant Name_Type:= "                    ";

  type Command_Type is (DEPART, ADD, CANCEL, FIND, LIST);
--
-- Packages required to manipulate Queues and Enumerated IO.
--
  package Name_String_Queue_Package is
        new Generic_Queue_Package
          (Item_Type => Name_Type);

  package Command_Type_IO is
        new Enumeration_IO (Command_Type);

  use Name_String_Queue_Package;
  use Command_Type_IO;

--
-- Variables
--
  Command: Command_Type;
  Name: Name_Type;
  Input_File: File_Type;
  Name_Queue: Queue_Type;
--
-- Get the names from the file
--
procedure Process_Name (Processed_File: in out File_Type;
                        Processed_Name: out Name_Type) is separate;
--
-- List the passengers in the queue
--
procedure List_Passengers (A_Queue: in Queue_Type) is separate;

begin -- Flights
  Open (Input_File, In_File, "Flight_Data.dat");
```

```
--
-- Processing begins. Use of case statements allows easy handling of
-- the different commands.
--
while not End_Of_File (Input_File) loop
  Name:= Blank_Name;

  begin -- Exception block for incorrect commands
    get (Input_File, Command);
   Put (Command);
    case Command is
      when DEPART => Put_Line ("Flight Departing");
                     Skip_line (Input_File);
      when ADD    => Put (Command);
                     Put (" ");
                     Process_Name (Input_File, Name);
                     Put (Name);
                     Put ("added to the Flight List.");
                     Enqueue (Name, Name_Queue);
                     New_Line;
      when CANCEL => Put (" ");
                     Process_Name (Input_File, Name);
                     Put (Name);
                     New_Line;
      when FIND   => Put (" ");
                     Process_Name (Input_File, Name);
                     Put (Name);
                     New_Line;
      when LIST => Put (command);
                     Skip_Line (Input_File);
                     List_Passengers (Name_Queue);
    end case;
    New_Line;
  exception
    when Data_Error =>
      Put ("ERRONEOUS COMMAND");
      Skip_Line (Input_File);
      New_Line;
  end; -- Exception block for incorrect commands
 end loop; -- while not End_Of_File
end Flights;
```

The bold-faced text in figure 11 indicates new code which was inserted for delivery three. Figure 12 contains **List_Passengers.ada.**

FIGURE 12. List_Passengers.ada

```
separate (Flights)
procedure List_Passengers (A_Queue: in Queue_Type) is
--
-- This procedure prints the names of the passengers in the queue.
--
-- INPUT: A_Queue.....Queue containing list of passengers.
-- OUTPUT:
--  Each passengers name will be printed on the terminal.
--
  Passenger_Name: Name_Type;
  Passenger_Queue: Queue_Type;

  begin -- List
    Passenger_Name:= Blank_Name;
    Passenger_Queue:= A_Queue;
    New_Line; New_Line;
    Put ("Passenger List");
    New_Line; New_Line;
    Dequeue_Loop: loop -- Empty the passenger list.
      begin -- Exception block for empty queue
        Dequeue (Passenger_Name, Passenger_Queue);
        Put (Passenger_Name);
        New_Line;
      exception
        when Empty_Queue_Error =>
          exit Dequeue_Loop;
      end; -- Exception block for empty queue
    end loop Dequeue_Loop;
  end List_Passengers;
```

Nothing is Perfect

Ada

The greatest difficulty students encounter when using Ada is determining the proper compilation order of the modules. The primary order of confusion is that the rules for subprogram and subunits do not seem (to students, at least) to be consistent. Subprograms must be compiled before the main program unit and subunits are compiled afterwards. When packages are employed the additional complexity of separate specifications and bodies confuse students even more. Drawing dependency diagrams alleviates this confusion somewhat, but nothing short of experience seems to be a good solution. Fortunately, students are not confronted with the more complexly organized programs until after they have become more mature.

Other problems with Ada stem from the richness of the language. Ada offers far too many options for certain constructs. The idea of providing named, as well as, positional parameters for subprogram parameters is excellent; but the "mix and match" options are not. This sort of "feature" tends

to make Ada a very difficult language to teach lower level/entry students. An instructor must very carefully introduce and discuss these options. I feel these options (and others) must be discussed if only to warn students of the potential problems they pose. Teaching a subset of Ada should not be considered. Too often students will accidentally and erroneously incorporate an untaught construct into their program and expend many hours attempting to isolate and correct a problem.

Incremental Program Development

There are two major problems with the use of IPD.

Instructors must realize that IPD will increase their workload. Program specifications and delivery requirements **must** be well thought out.Examining and grading the intermediate deliveries is a time consuming, but necessary, process. This method will not work unless the instructor is willing to expend the requisite extra effort.

If used too long, IPD can actually stymie student initiative and innovation. Students can become so reliant on the "prefabricated" delivery requirements that they their ability to design and implement a project unassisted can be seriously degraded. Present research indicates that this method is most beneficial starting with the third (of six programs) program assignment of CS 1 and should continue though program assignment four or five (of five to six programs) of CS 2. Assignments earlier than the third in CS 1 are usually too small to warrant the use of this method and at some point students must demonstrate the capability to "build" a complete program without assistance. Research is continuing in this area. IPD can and should be used somewhat in CS 5 and 7. In these courses it should not be used for all of the programs and the delivery requirements should be less detailed.

Synergy

The IPD paradigm is an extremely valuable tool and can be effectively used in combination with any language. Our department uses Pascal as its main language and I have used IPD successfully for a number of years.

Ada is an excellent language for software development. While it is not our main language, we have been offering at least one Ada course per semester for the past five years and use it extensively in our Programming Languages, Advanced Operating Systems, and Software Engineering courses. Our experience with Ada has led us to seriously consider incorporating it as the main language for the curriculum.

While IPD and Ada are each effective teaching tools, in their own right, the combination of the two creates an extremely powerful and potent instrument for the instruction of lower level Computer Science students. The effect is synergetic, where the requirements of IPD (modularity, elegant transition from design to code, and incremental delivery) are amply met by the features of Ada, resulting in a mechanism which **practically** illustrates and **consistently** reinforces the power of top-down design and step-wise refinement.

Student Reaction

At first student reaction is almost universally strongly against this methodology. After the second program (using incremental program development) most students grudgedly accept it, and somewhere during the third program the "light bulb" switches on. It is at this stage that the difficulty of the problem is such that solving it is significantly beyond their intellectual capacity without the use of proper software development techniques. Programs at this stage begin to incorporate many disparate entities which have to be properly assembled in order to obtain a working program. It is in the assembly of these entities that students have problems and once they realize that incremental program development provides them with many of the tools to make their job easier, they accept it and use it.

The brightest students are usually the ones who most resist using this methodology. I believe that these students are not seriously challenged (by programs) until much later in their course work. It is still important for them to go through the process, because when they are faced with a problem which they can not solve "naturally" (and this is inevitable) they will have the tools to assist them.

Summary

Incremental program development is an important method when used to teach and illustrate important techniques in program development.It is especially effective when used in conjunction with the Ada programming language. Students acquire an appreciation and understanding of the power these techniques offer when they see how much easier it is to complete assignments. Preliminary indications are that it is most valuable when used in CS 1 and 2.

This methodology should be used with care, however. An instructor must specify intermediate results so as to reinforce class room discussion of proper program development procedures. The instructor must also be aware of when to discontinue its use. If used too long or incorrectly this technique could well become detrimental to student development

References

Booch, G., *Software Components with Ada*, Benjamin/Cummings, 1987.

Hoare, C. A. R., *Hints On Programming Language Design, Sigact/SigPlan Symposium of Principles of Programming Languages*, Octover 1973

Pratt, T. W., *Programming Languages: Design and Implementation*, 2d. ed., Prentice-Hall, 1984.

Pressman, R. S., *Software Engineering: A Practitioner's Approach*, 2d.ed., McGraw-Hill, 1987.

Schneider, M. G. and Bruell, B. C., *Advanced Programming and Problem Solving with Pascal*, 2d. ed., John Wiley, 1987.

Sebesta, R. W., *Concepts of Programming Languages*, Benjamin/Cummings, 1989.

Wirth, N., *On the Design of Programming Languages, Proceedings IFIP CONGRESS 74*, 386-393, North-Holland Publishing Company.

This Page Left Blank Intentionally

USING SHORT LABORATORY EXERCISES TO DEVELOP Ada AWARENESS

Patricia K. Lawlis
Dept. of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Martin R. Stytz
Dept. of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

I. INTRODUCTION.

Students can learn the Ada programming language in many ways, with the approach heavily influencing the student's perspective on the language. The use of traditional, syntax-directed, bottom-up teaching methodologies develops students into traditional programmers. They learn programming-in-the-small, and Ada is perceived as just another language, albeit with some unique constructs that support object-oriented programming. On the other hand, a top-down approach to teaching Ada introduces students to the language as a software engineering tool. In this new approach, students begin with an introduction to the significant Ada constructs and their relationship to the goals of object-oriented programming. Instead of being taught Ada syntax, it is assimilated through repeated exposure to the language within a controlled laboratory learning environment. Rather than doing simple "Hello world!" programs from scratch, the students are introduced to Ada-based software engineering through exposure to software engineering concepts in class and implementations of the same concepts in lab. To succeed in this top-down teaching style, students must build on given code so that they can avoid the learning barriers encountered when programming-in-the-small and building systems from scratch.

This paper describes five short, independent laboratory exercises emphasizing features of Ada which support software engineering concepts. The exercises are used in a software design course that requires students to use Ada for system prototyping and design. The students in the course have a wide range of prior programming experience, but most have little prior exposure to Ada. Each exercise emphasizes a single Ada feature and the support of the feature for a general principle of software engineering. Hence, some or all of the labs could be used in courses other than one that deals strictly with software design.

Section 2 of this paper discusses the philosophy behind the development and use of these exercises. Section 3 discusses the exercises in detail, and Section 4 summarizes the results of using them. The appendix contains the exercises along with the required Ada code.

II. PHILOSOPHY BEHIND THE EXERCISES.

These five exercises exploit two important aspects of learning. People learn by doing and they learn by example. Programming exercises always provide an opportunity for students to learn by doing. However, the initial exposure to the language can be a very frustrating experience unless it is appropriately controlled. We control the learning environment by providing Ada source code that the students use to complete the labs. The use of given code achieves two important objectives. First, the labs provide an example of good Ada coding style, which the students are encouraged to adopt for their own use. Second, given code reduces the amount of information the students must assimilate to implement a running program, which decreases student frustration in learning the language.

Three of these exercises provide students with an opportunity to learn by doing while the given code provides an example of Ada syntax. In these exercises, the students

begin the lab with working program modules, and the lab requires them to modify the modules to meet new design specifications. Hence, they have working examples to use as a basis for future work, and the exercise provides insights into the effects of specific Ada language constructs. In addition, the step-by-step directions for performing the lab eliminates confusion concerning lab goals and expected student performance.

The other two exercises provide students with an opportunity to investigate the issues involved in going from a software design to an Ada implementation of the design. In these exercises, the students start with either Booch diagrams or structure charts and must develop the relevant package specifications and stubbed modules.

III. THE EXERCISES.

1. Fundamentals

This lab introduces the Ada package as the basic language construct. It also exposes students to the basic Ada syntax. The lab employs the Verdix Ada Development System (VADS) on Unix, but that could easily be modified to accommodate a different Ada platform. Note that the specification and the body of the *Fractions* package are put in separate files to introduce students to this standard way of organizing a package. Also note that the Test_Fractions procedure is not very sophisticated, but the exercise provides a relatively simple example of Ada syntax which the students can grasp quickly.

2. Object-oriented design

This lab introduces the student to the software engineering process of mapping a design from a Booch diagram to Ada package specifications. Any simple Booch diagram can be used, but it should probably contain only two packages, each of which contain no more than six operations in the specification. Although the lack of an accompanying written specification of the packages frustrates the students, we feel omitting the specification forces the students to concentrate on the purpose of the lab. We have found that as soon as students see a written specification they begin to develop a complete design and working program. These efforts do not support the intent of the lab. By providing only a graphic design, students are forced to concentrate on translating the diagram into package specifications and developing their understanding of this important process.

3. Functional design

This lab introduces the student to the software engineering process of mapping a design from a Structure Chart to nested Ada procedures and functions. The *separate* clause, and its effect upon compilation order, is introduced in this context. We provide a structure chart that addresses the same design problem addressed in the object-oriented design lab. A diagram is provided instead of a complete specification for the same reason that only a Booch diagram is furnished in the object-oriented design lab. However, we do provide the students with an example of the use of the separate clause.

4. Package interfaces

This lab highlights the differing effects of the regular type, private type, and limited private type declaration when encapsulating an abstract data type (ADT) in an Ada package. This lab is an involved exercise, requiring the students to follow the lab directions very carefully. The lab has been simplified as much as possible, with most student actions consisting of changing statements to and from Ada comments. Note that in this lab we have put both the specification and the body of package *Fractions* in one file, illustrating

another option for package organization and simplifying the compilation process required by the exercise.

5. Exception handling

This lab allows students to explore the Ada exception handling and propagation mechanisms. In the process, it also helps to clarify issues of scope. This lab is a somewhat contrived example of exception propagation, but its deliberate simplicity makes the code and exception propagation easy to understand

IV. RESULTS.

These exercises have already been modified based on feedback from students who have done them,and they can undoubtedly be further improved. They could also be tailored to different situations and different courses.

We have noted two important results from their use. First, students almost always complete them, so the exercises are neither threatening nor an area of potential failure. Second, students who have completed the labs almost unanimously agree that they learned a lot about Ada from them. Their demonstrated ability to use Ada and software engineering design techniques on a larger course project supports this conclusion.

V. APPENDIX.

Style conventions used in the Ada code for laboratory exercises 1, 4, and 5:
Reserved words are in **boldface.**
Package names are in *italics.*
User-defined identifiers have the first letter Capitalized.
Standard Ada package names are in *ALL-CAPITALS.*

LABORATORY 1
FUNDAMENTALS OF UNIX AND ADA

Objectives:

1. To become familiar with Unix.
2. To become familiar with the Verdix Ada Development System (VADS).
3. To learn fundamental Ada syntax by reading and modifying code.

Procedure:

1. Have pencil and paper at hand, and take notes as you work through this lab. Note the commands you use as well as any noteworthy results or insights. You need not recopy the test input or output data into your notes, but you should note whether or not the output was correct for each test case.

2. Login. Create a directory called lab for the lab work of this course, using the following command:

```
mkdir lab
```

   Then protect the directory by issuing the following command:

```
chmod 700 lab
```

   Check the directory permissions by typing:

```
ls -l
```

   What permissions are now associated with the directory lab?

3. Make lab your current directory with the command:

```
cd lab
```

   Now create a VADS directory (program library directory) for this lab exercise using the following command:

```
a.mklib lab1
```

4. Copy both the specification and the body of the *Fractions* package as well as the test program **Test_Fractions** into your lab1 directory using the following commands. Don't forget the dots (.).

```
cd lab1
cp <course directory>/LAB/Fractions.a .
cp <course directory>/LAB/FractionsBody.a .
cp <course directory>/LAB/Test_Fractions.a .
```

5. Compile the specification of the *Fractions* package using the command:

```
ada Fractions.a
```

   Compile the body of *Fractions* with:

```
ada FractionsBody.a
```

6. Compile the **Test_Fractions** program using the following command:

```
ada -M Test_Fractions.a
```

and run the program using the following command:

```
a.out
```

and the following test data:

X	Y
N = 6, D = 8	N = 3, D = 4
N = 3, D = 1	N = -7, D = -7
N = 0, D = 4	N = 1, D = 150
N = 15000, D = 1	N = 1, D = 15000
N = 4, D = 0	N = 5, D = 1
N = 1.0, D = 5	N = 5, D = 1
N = yes, D = 5	N = 5, D = 1

7. In package *Fractions*, where type **Fraction** is declared as a private type (in the specification file, Fractions.a), change the type of the variable **Numerator** from **integer** to **float** (and its initial value from 0 to 0.0). This change necessitates modifications to the package body. Note that these changes do not propagate to the package specifications.

   Make the necessary changes to the package body as follows:

   In function Top, the line

```
return X.Numerator;
```

   must be changed to read

```
return integer ( X.Numerator );
```

   In the definition of the function called **Make_Fraction**, the line

```
then return ( Top, Bottom );
```

   must be changed to read:

```
then return ( float ( Top ), Bottom );
```

   and the following line must be changed to read:

```
else return ( float ( -Top ), -Bottom );
```

8. Recompile both the specification and body of package *Fractions*. Note that the specification must be recompiled only because something is changed in its private part, and <u>not</u> because any of the interfaces to the package have changed.

9. Recompile and run **Test_Fractions** on the test data. Note that the user interface with the program hasn't changed -- integers are still entered for the fraction numerators. The changes are implementation changes only, invisible to the package user.

10. Submit a listing of your updated Fractions package, together with your notes on what occurred as you progressed through this lab and your interpretation thereof.

```
-------------------------------------------------------------------------------
--                        PACKAGE FRACTIONS                                  --
--                          SPECIFICATION                                    --
-------------------------------------------------------------------------------

--    THIS PACKAGE CONTAINS THE DEFINITION FOR AN ABSTRACT FRACTION TYPE. THIS  --
--     INCLUDES ARITHMETIC AND RELATIONAL OPERATIONS FOR A FRACTION. IT ALSO    --
--    INCLUDES THE CAPABILITY TO CREATE AND REDUCE A FRACTION AS WELL AS TO GET --
                          AND PUT ONE INTERACTIVELY.
-------------------------------------------------------------------------------
package Fractions is
        type Fraction is private;

        function Make_Fraction ( Top, Bottom : in integer ) return Fraction;
        function Reduce      ( X : in Fraction ) return Fraction;
        function Equal       ( Left, Right : in Fraction ) return boolean;
        function "<"         ( Left, Right : in Fraction ) return boolean;
        function "<="        ( Left, Right : in Fraction ) return boolean;
        function ">"         ( Left, Right : in Fraction ) return boolean;
        function ">="        ( Left, Right : in Fraction ) return boolean;
        function "+"         ( Left, Right : in Fraction ) return Fraction;
        function "-".        ( Left, Right : in Fraction ) return Fraction;
        function "*"         ( Left, Right : in Fraction ) return Fraction;
        function "/".        ( Left, Right : in Fraction ) return Fraction;

        procedure Get        ( X : out Fraction );
        procedure Put        ( X : in Fraction );

private
    type Fraction is
        record
            Numerator   : integer := 0;
            Denominator : positive := 1;
        end record;
end Fractions;
```

```
--------------------------------------------------------------------------------

--                          P A C K A G E   F R A C T I O N S                 --

--                                    BODY                                    --
--------------------------------------------------------------------------------

--    THIS PACKAGE CONTAINS THE DEFINITION FOR AN ABSTRACT FRACTION TYPE. THIS  --
--      INCLUDES ARITHMETIC AND RELATIONAL OPERATIONS FOR A FRACTION. IT ALSO   --
--    INCLUDES THE CAPABILITY TO CREATE AND REDUCE A FRACTION AS WELL AS TO GET  --
--                    AND PUT ONE INTERACTIVELY.                              --
--------------------------------------------------------------------------------

with TEXT_IO;
package body Fractions is
    package Int_IO is new TEXT_IO.INTEGER_IO (integer);
--------------------------------------------------------------------------------

--                              FUNCTION TOP                                  --

--    THIS FUNCTION RETURNS THE VALUE OF THE NUMERATOR OF THE GIVEN FRACTION.   --
--------------------------------------------------------------------------------
function Top ( X : in Fraction ) return integer is
    begin
        return X.Numerator;
    end Top;
--------------------------------------------------------------------------------

--                            FUNCTION BOTTOM                                 --

--   THIS FUNCTION RETURNS THE VALUE OF THE DENOMINATOR OF THE GIVEN FRACTION.  --
--------------------------------------------------------------------------------
function Bottom ( X : in Fraction ) return positive is
    begin
        return X.Denominator;
    end Bottom;
--------------------------------------------------------------------------------

--                          FUNCTION MAKE_FRACTION                            --

--     THIS FUNCTION RETURNS A FRACTION COMPOSED OF THE GIVEN NUMERATOR AND    --
--                                DENOMINATOR.                                --
--                                                                            --
--------------------------------------------------------------------------------
function Make_Fraction ( Top, Bottom : in integer ) return Fraction is
    begin
        if Bottom > 0 then return ( Top, Bottom );
        else return ( -Top, -Bottom );
        end if;
    end Make_Fraction;
--------------------------------------------------------------------------------

--                            FUNCTION REDUCE                                 --

--          THIS FUNCTION RETURNS THE REDUCED FORM OF THE GIVEN FRACTION.     --
--------------------------------------------------------------------------------
function Reduce ( X : in Fraction ) return Fraction is
```

```
    The_GCD,
    New_Top,
    New_Bottom : integer;
-------------------------------------------------------------------------------------------------------
--                          ---------INTERNAL FUNCTION GCD ---------                          --
--     THIS FUNCTION RETURNS THE GREATEST COMMON DIVISOR OF THE GIVEN NUMBERS      --
-------------------------------------------------------------------------------------------------------
function GCD ( First , Second : in integer ) return integer is
    begin
        if ( First = 0 or Second = 0 ) then return 1;
            elsif First < Second then return GCD ( Second , First );
            elsif ( Second <= First ) and ( First mod Second = 0 ) then return Second;
            else return GCD ( Second , First mod Second );
        end if;
    end GCD;
-------------------------------------------------------------------------------------------------------
    begin                    -- body of function Reduce
        The_GCD := GCD ( abs Top ( X ) , abs Bottom ( X ) );
        New_Top := Top ( X ) / The_GCD;
        New_Bottom := Bottom ( X ) / The_GCD;
        return Make_Fraction ( New_Top , New_Bottom );
    end Reduce;
-------------------------------------------------------------------------------------------------------
--                                  FUNCTION "="                                  --
--   THIS FUNCTION RETURNS TRUE IF THE GIVEN FRACTIONS ARE EQUIVALENT, AND FALSE    --
--                                  OTHERWISE.                                    --
-------------------------------------------------------------------------------------------------------
function Equal ( Left , Right : in Fraction ) return boolean is
    begin
        return ( Top ( Left ) * Bottom ( Right ) = Top ( Right ) * Bottom ( Left ) );
    end Equal;
-------------------------------------------------------------------------------------------------------
--                                  FUNCTION "<"                                  --
--       THIS FUNCTION RETURNS TRUE IF THE LEFT FRACTION IS LESS THAN THE RIGHT     --
--                          FRACTION, AND FALSE OTHERWISE.                        --
-------------------------------------------------------------------------------------------------------
function "<" ( Left , Right : in Fraction ) return boolean is
    begin
        return ( Top ( Left ) * Bottom ( Right ) < Top ( Right ) * Bottom ( Left ) );
    end "<";
-------------------------------------------------------------------------------------------------------
--                                  FUNCTION "<="                                 --
--   THIS FUNCTION RETURNS TRUE IF THE LEFT FRACTION IS LESS THAN OR EQUAL TO THE  --
--                      RIGHT FRACTION, AND FALSE OTHERWISE                       --
-------------------------------------------------------------------------------------------------------
function "<=" ( Left , Right : in Fraction ) return boolean is
    begin
        return ( Top ( Left ) * Bottom ( Right ) <= Top ( Right ) * Bottom ( Left ) );
    end "<=";
```

```
-------------------------------------------------------------------------------------------------------

--                                        FUNCTION ">"                                               --

--     THIS FUNCTION RETURNS TRUE IF THE LEFT FRACTION IS GREATER THAN THE RIGHT                      --
--                          FRACTION, AND FALSE OTHERWISE.                                            --

-------------------------------------------------------------------------------------------------------
function ">" ( Left , Right : in Fraction ) return boolean is
    begin
        return ( Top ( Left ) * Bottom ( Right ) > Top ( Right ) * Bottom ( Left ) );
    end ">";
-------------------------------------------------------------------------------------------------------

--                                        FUNCTION ">="                                              --

--   THIS FUNCTION RETURNS TRUE IF LEFT FRACTION IS GREATER THAN OR EQUAL TO THE                      --
--                          RIGHT FRACTION, AND FALSE OTHERWISE                                       --

-------------------------------------------------------------------------------------------------------
function ">=" ( Left , Right : in Fraction ) return boolean is
    begin
        return ( Top ( Left ) * Bottom ( Right ) >= Top ( Right ) * Bottom ( Left ) );
    end ">=";
-------------------------------------------------------------------------------------------------------

--                                        FUNCTION "+"                                               --

--              THIS FUNCTION RETURNS THE SUM OF THE GIVEN FRACTIONS.                                 --

-------------------------------------------------------------------------------------------------------
function "+" ( Left, Right : in Fraction ) return Fraction is
    New_Top , New_Bottom : integer;
    Result : Fraction;

    begin
        New_Top := ( Top ( Left ) * Bottom ( Right ) ) + ( Top ( Right ) * Bottom ( Left ) );
        New_Bottom := Bottom ( Left ) * Bottom ( Right );
        Result := Reduce ( Make_Fraction (New_Top, New_Bottom) );
        return Result;
    end "+";
-------------------------------------------------------------------------------------------------------

--                                        FUNCTION "-"                                               --

--              THIS FUNCTION RETURNS THE DIFFERENCE OF THE GIVEN  FRACTIONS.                         --

-------------------------------------------------------------------------------------------------------
function "-" ( Left, Right : in Fraction ) return Fraction is
    New_Top , New_Bottom : integer;
    Result : Fraction;

    begin
        New_Top := ( Top ( Left ) * Bottom ( Right ) ) - ( Top ( Right ) * Bottom ( Left ) );
        New_Bottom := Bottom ( Left ) * Bottom ( Right );
        Result := Reduce ( Make_Fraction (New_Top, New_Bottom) );
        return Result;
    end "-";
```

```
-------------------------------------------------------------------------------------------------------

--                                    FUNCTION "*"                                                  --

--          THIS FUNCTION RETURNS THE PRODUCT OF THE GIVEN  FRACTIONS.                              --
-------------------------------------------------------------------------------------------------------
function "*" ( Left, Right : in Fraction ) return Fraction is
    New_Top , New_Bottom : integer;
    Result : Fraction;

    begin
        New_Top := Top ( Left ) * Top ( Right );
        New_Bottom := Bottom ( Left ) * Bottom ( Right );
        Result  := Reduce ( Make_Fraction (New_Top, New_Bottom) );
        return Result;
    end "*";
-------------------------------------------------------------------------------------------------------

--                                    FUNCTION "/"                                                  --

--          THIS FUNCTION RETURNS THE QUOTIENT OF THE GIVEN  FRACTIONS.                             --
-------------------------------------------------------------------------------------------------------
function "/" ( Left, Right : in Fraction ) return Fraction is
    New_Top , New_Bottom : integer;
    Result : Fraction;

    begin
        New_Top := Top ( Left ) * Bottom ( Right );
        New_Bottom := Bottom ( Left ) * Top ( Right );
        Result := Reduce ( Make_Fraction (New_Top, New_Bottom) );
        return Result;
    end "/";
-------------------------------------------------------------------------------------------------------

--                                    PROCEDURE GET                                                 --

--  THIS PROCEDURE GETS A FRACTION AS INPUT FROM THE KEYBOARD AND THEN RETURNS                        --
--                                IT TO THE CALLING PROCEDURE.                                      --
-------------------------------------------------------------------------------------------------------
procedure Get ( X : out Fraction ) is
    Top , Bottom : integer;

    begin
        Int_IO.Get ( Top );
        Int_IO.Get ( Bottom );
        X := Make_Fraction ( Top , Bottom );
    end Get;
-------------------------------------------------------------------------------------------------------

--                                    PROCEDURE PUT                                                 --

--          THIS PROCEDURE PUTS THE GIVEN FRACTION TO THE SCREEN                                    --
-------------------------------------------------------------------------------------------------------
procedure Put ( X : in Fraction ) is
    begin
        Int_IO.Put ( Top ( X ) );
        TEXT_IO.PUT ( " / " );
        Int_IO.Put ( Bottom ( X ) );
```

```
    end Put;

end Fractions;
```

```
-------------------------------------------------------------------------------------------------
--                                    T E S T P R O G R A M                                    --
--                                    T E S T _ F R A C T I O N S                              --
-------------------------------------------------------------------------------------------------
--              THIS PROGRAM TESTS THE CAPABILITIES OF PACKAGE FRACTIONS                       --
-------------------------------------------------------------------------------------------------
with TEXT_IO, Fractions;
use TEXT_IO, Fractions;          --Use clause necessary for exceptions and infix operations

procedure Test_Fractions is
    X , Y: Fraction;
    Continue : character := 'n';          -- indicates whether to continue or to quit

    begin
        loop            --Processes any number of fractions
            begin                   -- block required for exception handlers
                        -- ********************************************
                        -- Get the fractions (not a sophisticated way of doing this)
                        -- ********************************************
                TEXT_IO.NEW_LINE;
                TEXT_IO.PUT_LINE ( "Please enter a fraction X");
                TEXT_IO.PUT_LINE ( "Enter the fraction as ""N D""" );
                TEXT_IO.PUT_LINE ( "    where N is the numerator and D is the denominator" );
                Fractions.Get ( X );
                TEXT_IO.NEW_LINE;
                TEXT_IO.PUT_LINE ( "Please enter a second fraction Y in the same manner" );
                Fractions.Get ( Y );
                TEXT_IO.NEW_LINE;
                TEXT_IO.PUT ( "X = " );
                Fractions.Put ( X );
                TEXT_IO.PUT ( "        " );
                TEXT_IO.PUT ( "Y = " );
                Fractions.Put ( Y );
                TEXT_IO.NEW_LINE;

                        -- **************************
                        -- Reduce one fraction
                        -- **************************
                TEXT_IO.PUT ( "X in lowest terms is " );
                Fractions.Put ( Reduce ( X ) );
                TEXT_IO.NEW_LINE;

                        -- **************************
                        -- Test the relational operations
                        -- **************************
                if X = Y then TEXT_IO.PUT ( "X = Y" );
                    else TEXT_IO.PUT ( "X not = Y" );
                end if;
```

```
            TEXT_IO.NEW_LINE;
            if X < Y then TEXT_IO.PUT ( "X < Y" );
                else TEXT_IO.PUT ( "X not < Y" );
            end if;
            TEXT_IO.NEW_LINE;
            if X >= Y then TEXT_IO.PUT ( "X >= Y" );
                else TEXT_IO.PUT ( "X not >= Y" );
            end if;
            TEXT_IO.NEW_LINE;
            if X <= Y then TEXT_IO.PUT ( "X <= Y" );
                else TEXT_IO.PUT ( "X not <= Y" );
            end if;
            TEXT_IO.NEW_LINE;
            if X > Y then TEXT_IO.PUT ( "X > Y" );
                else TEXT_IO.PUT ( "X not > Y" );
            end if;
            TEXT_IO.NEW_LINE;

                -- ***************************
                -- Test the arithmetic operations
                -- ***************************
            TEXT_IO.PUT ( "X + Y = " );
            Fractions.Put ( X + Y );
            TEXT_IO.NEW_LINE;
            TEXT_IO.PUT ( "X - Y = " );
            Fractions.Put ( X - Y );
            TEXT_IO.NEW_LINE;
            TEXT_IO.PUT ( "X * Y = " );
            Fractions.Put ( X * Y );
            TEXT_IO.NEW_LINE;
            TEXT_IO.PUT ( "X / Y = " );
            Fractions.Put ( X / Y );
            TEXT_IO.NEW_LINE;
            TEXT_IO.PUT ( "( X + Y ) * ( X - Y ) = " );
            Fractions.Put ( ( X + Y ) * ( X - Y ) );
            TEXT_IO.NEW_LINE; TEXT_IO.NEW_LINE;
            TEXT_IO.PUT ( "Do you wish to continue? (Y or N) " );
            TEXT_IO.GET ( Continue );
            if Continue = 'n' or Continue = 'N' then exit;
            end if;

            exception
                when DATA_ERROR =>
                                    TEXT_IO.PUT_LINE ("Invalid input, please try again: ");
                TEXT_IO.SKIP_LINE;    -- required to clear buffer
        end;          -- End exception handler block
    end loop;

    TEXT_IO.PUT_LINE ( "Test over" );

    exception
        when others => TEXT_IO.PUT_LINE ( "Unknown error occurred" );
        raise;        -- Allows system message to indicate the name of raised exception

end Test_Fractions;
```

LABORATORY 2
OBJECT ORIENTED DESIGN

Objectives:

1. To gain experience with the object-oriented design (OOD) process.
2. To practice developing Ada specifications from Booch diagrams.

Procedure:

1. Have pencil and paper at hand, and take notes as you work through this lab. Be sure to indicate any noteworthy results or insights.

2. Develop Ada package specifications (only the specs, and only for the two packages -- not for the "Driver") for the cruise control problem, using the attached Booch graph as the basis for developing these specs. Include what you think would be appropriate calling parameters for the procedures and functions (make them up, but be sure to use some parameters -- some in, some out, and some both -- on at least half of the procedures and functions). Be sure to comment appropriately.

3. Compile the specs you develop. Be sure they compile without error. (Do not attempt to execute the code -- specs cannot be executed.)

LABORATORY 3
FUNCTIONAL DESIGN LAB

Objectives:

1. To gain experience with the functional design process.
2. To practice developing Ada specifications from structure charts.

Procedure:

1. Have pencil and paper at hand, and take notes as you work through this lab. Be sure to indicate any noteworthy results or insights.

2. Develop stubbed Ada code for the cruise control problem, using the attached structure chart as the basis for developing this code. In the top level procedure, specify each lower level procedure or function with a **separate** clause. Then create these separate procedures and/or functions in separate files. Include calling parameters for procedures and functions as in the previous lab. Also, include calls to the lower level procedures and functions in the top level procedure, and include at least one output statement in each lower level procedure. Be sure to comment appropriately.

3. Compile the code you develop. Each file may be compiled separately to check syntax. When it comes time to put them all together, you can create an executable file using the command:

   ada -M Cruise_Control_System [filename1] [filename2] ... [filenameN]

   where the file for the main procedure is filename1 and each separate file you have created is listed as one of the other filenames. Execute your program shell.

LABORATORY 4
PACKAGE INTERFACES LAB

Objectives:

1. To learn the differences in the use of an Ada package which depend on the declaration of the object type as a regular type, a private type, or a limited private type.

Procedure:

1. Have pencil and paper at hand, and take notes as you work through this lab. Be sure your notes describe the results of each step of this procedure, as well as your understanding of what is happening.

2. Copy the *Fractions* package and test program into your VADS directory using appropriate commands. The files are located in the directory

   <course directory>

   and the file names are Fractions.a and Test_Private.a

3. Compile both files. If any lines of **Test_Private** won't compile, comment them out by inserting 2 dashes at the beginning of each such line. Create an executable file called Regular_Type.

4. Run Regular_Type.

5. Change the declaration of type **Fraction** to a **private** type. Take out all dashes that you added to both files in step 3, if any (but be sure to leave in any that were there in the first place). Compile both files again. Again, comment out any lines which won't compile in the same manner as before. Create an executable file called Private_Type.

6. Run Private_Type.

7. Change the declaration of type **Fraction** to a **limited private** type. Take out all dashes that you added to both files in step 5, if any. Compile both files again, and again if any lines in **Test_Private** won't compile, comment them out as before, but don't make an executable file yet.

8. Remove the dashes from in front of the two new **Make_Fraction** calls in **Test_Private.** Also, comment out the specification for function **Make_Fraction** in package ***Fractions***. Then take out the dashes in front of the specification for procedure **Make_Fraction.** Create a procedure **Make_Fraction** in the body of package ***Fractions*** so it will work using the parameters given in the specification. Compile both files again, and create an executable file called Limited_Private_Type.

9. Run Limited_Private_Type and observe what happens.

```
--------------------------------------------------------------------------------
--                        P A C K A G E   F R A C T I O N S                   --
--                                SPECIFICATION                               --
--------------------------------------------------------------------------------

--    THIS PACKAGE CONTAINS THE DEFINITION FOR AN ABSTRACT FRACTION TYPE. THIS  --
--    INCLUDES ARITHMETIC AND RELATIONAL OPERATIONS FOR A FRACTION. IT ALSO     --
--    INCLUDES THE CAPABILITY TO CREATE AND REDUCE A FRACTION AS WELL AS TO GET --
--                    AND PUT ONE INTERACTIVELY.                                --
--------------------------------------------------------------------------------
package Fractions is
    type Fraction is
        record
            Numerator   : integer := 0;
            Denominator : positive := 1;
        end record;

-       procedure Make_Fraction (   Top, Bottom : in integer;
                                   Result       : out Fraction);
        function Make_Fraction ( Top, Bottom : in integer ) return Fraction;
        function Reduce     ( X : in Fraction ) return Fraction;

        function Equal      ( Left, Right : in Fraction ) return boolean;
        function "<"        ( Left, Right : in Fraction ) return boolean;
        function "<="       ( Left, Right : in Fraction ) return boolean;
        function ">"        ( Left, Right : in Fraction ) return boolean;
        function ">="       ( Left, Right : in Fraction ) return boolean;
        function "+"        ( Left, Right : in Fraction ) return Fraction;
        function "-".       ( Left, Right : in Fraction ) return Fraction;
        function "*"        ( Left, Right : in Fraction ) return Fraction;
        function "/".       ( Left, Right : in Fraction ) return Fraction;

        procedure Get       ( X : out Fraction );
        procedure Put       ( X : in Fraction );
end Fractions;


--------------------------------------------------------------------------------
--                        P A C K A G E   F R A C T I O N S                   --
--                                BODY                                        --
--------------------------------------------------------------------------------
  --CODE FOR FRACTIONS PACKAGE BODY IS THE SAME AS THAT FOR THE FUNDAMENTALS LAB
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--                          P R O C E D U R E                                 --
--                          TEST_PRIVATE                                      --
--------------------------------------------------------------------------------
--      A PROGRAM FOR TESTING THE TYPE FRACTION WHEN DECLARED AS A REGULAR, A  --
--                      PRIVATE, AND A LIMITED PRIVATE TYPE.                   --
--------------------------------------------------------------------------------

with TEXT_IO;
with Fractions;

procedure Test_Private is
    Fraction_One,
    Fraction_Two,
    Fraction_Three : Fractions.Fraction;

    begin
        Fraction_One.Numerator := 3;
        Fraction_One.Denominator := 4;
        TEXT_IO.PUT ("Created new fraction ");
        Fractions.Put (Fraction_One);
        TEXT_IO.PUT_LINE (" ");
        TEXT_IO.PUT_LINE ("Used the fraction representation directly");

        Fraction_Two := Fractions.Make_Fraction (Top => 2, Bottom => 8);
        Fraction_Three := Fractions.Make_Fraction (Bottom => 4, Top => 1);
-       Fractions.Make_Fraction (Top => 2, Bottom => 8, Result => Fraction_Two);
-       Fractions.Make_Fraction (Result => Fraction_Three, Top => 1, Bottom => 4);

        if Fraction_Two = Fraction_Three
            then TEXT_IO.PUT_LINE ("The fractions tested equal");
        else TEXT_IO.PUT_LINE ("The fractions tested unequal");
        end if;

    exception
        when others => TEXT_IO.PUT_LINE ("Something went wrong");
        raise;
end Test_Private;
```

LABORATORY 5
EXCEPTION HANDLING LAB

Objectives:

1. To provide hands on experience in handling Ada exceptions.
2. To demonstrate exception propagation.
3. To gain additional insight into scoping concepts.

Procedure:

1. Have pencil and paper at hand, and take notes as you work through this lab. Be sure to indicate any noteworthy results or insights.

2. Copy the files to be used for this exercise into your VADS directory using appropriate commands. The files are located in the directory

   ```
   <course directory>
   ```

   and the file names are Propagate.a and Resources.a

3. Compile each file by typing the following commands:

   ```
   ada Resources.a
   ada -M Propagate.a
   ```

4. Execute the program, and note the order of the messages which print out, especially that the exception propagation messages occur in the reverse order of the procedure calls.

5. Change the **when others** to **when Crazy** in procedure **Not_in_Package**, and try to recompile Propagate.a. Leave the last change in place, and then move the declaration of **Crazy** to the specification of package *Container* and try compiling Propagate.a again. Why doesn't this work? Now move the declaration of **Crazy** to procedure **Propagate** and try compiling Propagate.a again. This time run the program. Now **Crazy** is visible to **Not_in_Package**, and the program runs as before.

6. To procedure **Raises_Exception**, add an exception handler for **Crazy** which produces a meaningful message and also contains a **raise** statement to re-raise the exception. Then recompile Resources.a (only necessary if you changed it), recompile Propagate.a, and run the program again. This illustrates that the exception began its propagation from **Raises_Exception**.

```
--------------------------------------------------------------------------------
--                              PROCEDURE                                     --
--                              PROPAGATE                                     --
--------------------------------------------------------------------------------
--          THIS PROCEDURE DEMONSTRATES THE PATH OF EXCEPTION PROPAGATION.     --
--------------------------------------------------------------------------------
with Resources;

procedure Propagate is

--------------------------------------------------------------------------------
--      THIS IS THE SPECIFICATION OF THE PACKAGE WHICH BOTH RAISES THE EXCEPTION --
--                          INITIALLY AND HAS THE EXCEPTION HANDLER            --
--------------------------------------------------------------------------------
package Container is
    procedure Has_Handler;
    procedure Raises_Exception;
end Container;

--------------------------------------------------------------------------------
--   THIS PROCEDURE IS NOT A PART OF PACKAGE CONTAINER, BUT CONTAINER IS VISIBLE --
--       TO IT, SO IT CALLS ONE OF CONTAINER'S PROCEDURES.  IT CANNOT HANDLE    --
--   CONTAINER'S EXCEPTION BY NAME, BUT IT CAN HANDLE IT WITH AN OTHERS CLAUSE, --
--                          AND IT CAN PROPAGATE IT.                           --
--------------------------------------------------------------------------------
procedure Not_in_Package is
    begin
        Resources.Not_in_Package_Message;
        Container.Raises_Exception;
        exception
            when others => Resources.Anonymous_Exception_Message;
            raise;
    end Not_in_Package;

--------------------------------------------------------------------------------
--                  THIS IS THE BODY OF THE PACKAGE SPECIFIED ABOVE            --
--------------------------------------------------------------------------------
package body Container is
    Crazy : exception;
            --*********************************
            --This procedure handles the exception
            --*********************************
    procedure Has_Handler is
        begin
            Resources.Has_Handler_Message;
            Not_in_Package;
            exception
                when Crazy => Resources.Tell_Everyone;
        end Has_Handler;
```

```
-------------------------------------------------------------------------------------------------------
--                    THIS PROCEDURE INITIALLY RAISES THE EXCEPTION                                  --
-------------------------------------------------------------------------------------------------------
procedure Raises_Exception is
    begin
        Resources.Raises_Exception_Message;
        raise Crazy;
    end Raises_Exception;
end Container;
-------------------------------------------------------------------------------------------------------
--                    THIS IS THE BODY OF THE MAIN PROPAGATE PROCEDURE                                --
-------------------------------------------------------------------------------------------------------
begin
    Resources.Propagate_Message;
    Container.Has_Handler;
end Propagate;
```

```
--------------------------------------------------------------------------------
--                              P A C K A G E                                 --
--                             RESOURCES                                      --
--------------------------------------------------------------------------------

--    THIS PACKAGE CONTAINS ALL OF THE TEXT MESSAGES REQUIRED BY THE EXCEPTIONS --
--                        EXAMPLE PROCEDURE PROPAGATE.                        --
--------------------------------------------------------------------------------

package Resources is
    procedure Tell_Everyone;
    procedure Not_in_Package_Message;
    procedure Anonymous_Exception_Message;
    procedure Has_Handler_Message;
    procedure Propagate_Message;
end Resources;

with TEXT_IO;

package body Resources is

procedure Tell_Everyone is
    begin
        TEXT_IO.PUT_LINE ("Crazy is handled by Has_Handler");
    end Tell_Everyone;
procedure Not_in_Package_Message is
    begin
        TEXT_IO.PUT_LINE ("Control goes to Not_in_Package");
    end Not_in_Package_Message;
procedure Anonymous_Exception_Message is
    begin
        TEXT_IO.PUT_LINE ("Exception is handled and reraised in Not_in_Package");
    end Anonymous_Exception_Message;
procedure Has_Handler_Message is
    begin
        TEXT_IO.PUT_LINE ("Control goes to Has_Handler");
    end Has_Handler_Message;
procedure Raises_Exception_Message is
    begin
        TEXT_IO.PUT_LINE ("Control goes to Raises_Exception, exception is raised");
    end Raises_Exception_Message;
procedure Propagate_Message is
    begin
        TEXT_IO.PUT_LINE ("Control starts at main procedure Propagate");
    end Propagate_Message;
end Resources;
```

Panel Session

Lessons Learned in Teaching Software Engineering with Ada

Charles Engle, Chair

This panel will discuss the lessons learned, summarized as advantages and disadvantages, from trying to introduce students to the concepts of software engineering using Ada as the language of implementation. All of the panel members have had "platform experience" and will relate the good, the bad, and the ugly side of this topic. For instance, how does the instructor give the student an understanding of the concept of abstract data types? In Ada the answer is usually a combination of the rich variety of data types and the concept of packages. But how does the instructor get over the hurdle of I/O in Ada and still require the student to define new types (no predefined numeric types allowed!)? And how does the instructor show the student how to avoid the "use" clause, yet still get infix visibility for overloaded operators? How does the concept of information hiding get across to the student when private types must be fully declared in the package specification (although in a private part)? What about Design aspects or requirements analysis, or specifications and PDL? What types of programming assignments in Ada have proven to be educational for the student, pedagogically satisfying for the instructor, yet still entertaining (Yes, entertaining!)? These and other questions will be answered by this panel as well as any questions from the audience.

This Page Left Blank Intentionally

Hard-Deadline, Soft-Deadline and No-Deadline Real-Time Programming: Telling the Truth when Teaching Tasking

Capt. David A. Cook
Dept. Of Computer Science
U.S. Air Force Academy, CO. 80840

Modern-day applications often have both real-time and parallel requirements. Just because Ada has *tasking*, it is sometimes erroneously assumed that Ada provides support for both real-time and parallel applications. This is not necessarily so. Ada does provide the minimal support necessary to allow experienced developers to design their own particular solutions to both of these types of problems, but the solutions are not inherently part of Ada itself. This article discusses the differences between the three various types of real-time applications, and also covers the various problems and solutions available, both within and external to Ada itself. Additionally, it covers why using Ada tasks as a "normal" programming practice can actually increase system response time. Explanations for various anomalies that tasking can produce are also mentioned. Finally, suggestions for proper use of tasking and situations where tasking is inappropriate are discussed.

Key terms and phrases: concurrent processing, deadlines, design, distributed processing, fault masking and fault-tolerance, parallel processing, real-time applications, recovery, rollbacks, tasking.

1. INTRODUCTION

Professional real-time software programmers often become irate when Ada programmers loudly profess that "Ada handles real-time programming because it has tasking". The truth of the matter is that educators often fail to adequately cover the basic terminology and

---

The author is currently a full-time Ph.D. student at Texas A&M University. Until May 1991, the author's address is 1403 Francis Drive, College Station, TX 77840.

Email address is dcook@ajpo.sei.cmu.edu

nomenclature required to discuss concurrent/parallel processing with an experienced real-time programmer.

Most experienced Ada educators would agree that a syntax-directed course is NOT the way to teach Ada. Instead, most agree that you should start with the concepts of abstraction and encapsulation, which in turn motivates the Ada packaging concept. Additionally, educators using a sound software engineering foundation teach that using "brute force" to tackle to problem is insufficient. Instead, they teach that a proper design is a necessity for solving anything other than a very small trivial problem. By teaching good design, educators provide students a good "conceptual framework" upon which to hang and organize more complex topics later.

The same treatment applies to teaching tasking. Tasking is much, much more than just breaking up an Ada system into large blocks than can somehow "execute concurrently" or even in "apparent parallelism". First of all, students (and software designers in general) need to establish when either parallel or concurrent programming is appropriate. It is fundamental that students see exactly when tasking can benefit the final product. Frederick Brooks says that there is an "inherent complexity" to programming [Brooks 1987], and this certainly applies to tasking. The trick is deciding upon just what the inherent complexity really is. Software designers must apply abstraction in the design so as to capture the inherent complexity, but abstract away the non-essential elements. I have had students write programs that required four generic instantiations and three tasks to implement a simple "Hello World" application. While it worked, it was far more complex than needed. If a complex design is not needed, simpler methods should suffice. When, then, does a task need to be part of the solution?

2. WHEN TASKING IS APPROPRIATE

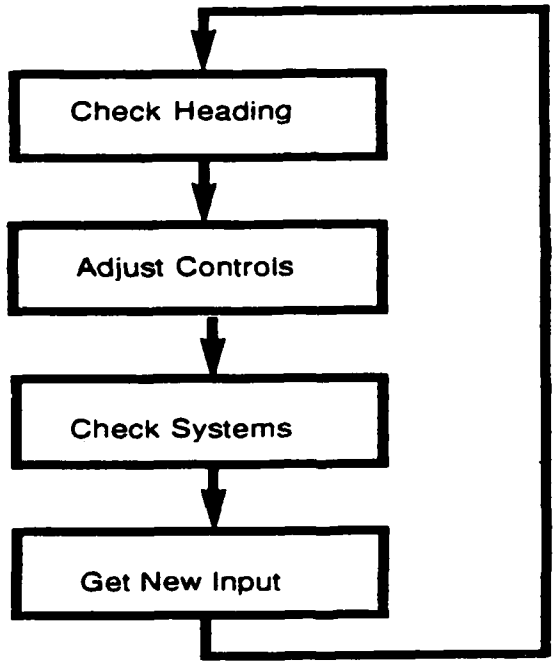
Often, educators brush off the question of when to use tasking by saying "Use tasking whenever parallelism is required". I doubt that many programmers (students and professionals alike) understand what parallelism actually is, let alone when it is required. In any case, it is wrong to let any student think that one of the prerequisites for using tasking is a problem that requires parallelism. Many sequential problems can be elegantly solved by correctly applying tasks, even though actual parallel processing will never be part of the solution.

2.1 The use of "Apparent Parallelism"

Few, if any educational Ada compilers actually support true parallel processing, anyway. Certainly, when using a PC-class machine (to include Macintosh machines, of course), a single CPU is all that is available. Large

systems also usually only have a single CPU (especially in an educational or development environment.) Even when using a machine like a Sequent (which can be configured for up to 10 processors), the single controller and dispatcher hosts all Ada tasks in a system upon a single processor. Educators often gloss over this, and continue to make sure that the students associate *Ada tasking* with *parallel processing.*

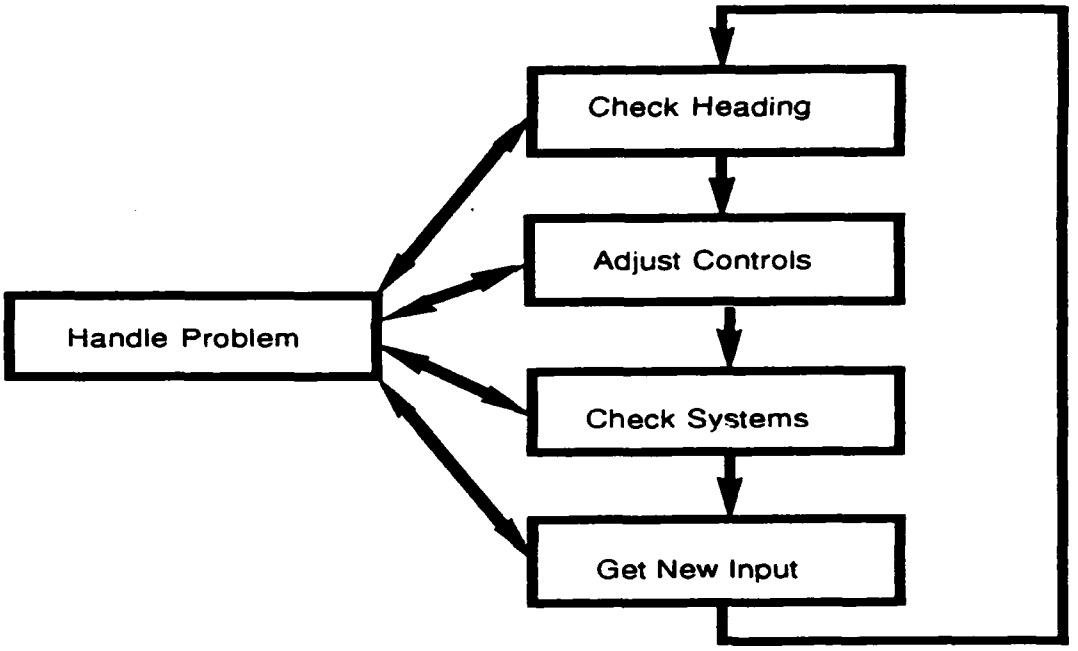
Granted, there is nothing wrong with using tasking on a single processor (*apparent parallelism* or *concurrent processing*) to provide an introduction or overview of parallel processing for a "cheap" price. However, students should be able to see that using tasking on a single-processor machine sometimes does have applications. As an example, imagine the problem of implementing a simple autopilot system. Functional modules might consist of the following:



Design #1

The sequential solution to the design above would follow easily. After implementation, I would now ask my students to consider how to implement a "error handling" or problem-handling procedure. Specifically, students should be able to write a single procedure that would handle errors that could occur in any of the other modules. Therefore, it should be able to both call and be called by all other "worker" program units.

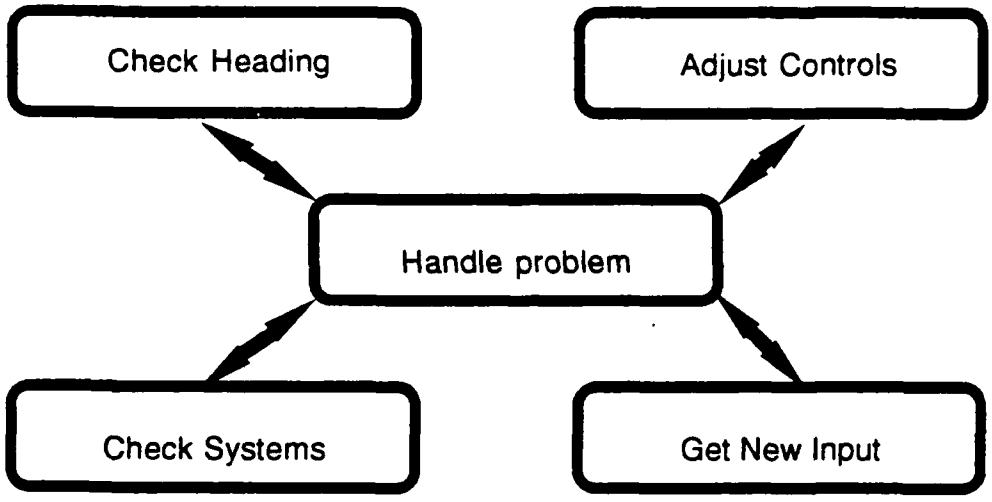
Although there certainly exist various ways of implementing the error-handling routine, the most easily visualizable design would now probably be:



Design #2

The current state of the design motivates a discussion as to which modules are more important. Does the program want to be "stuck" in "Get New Input" when there is a serious engine problem? Can it afford to wait until it "gets back around" to "Check Systems"? Probably not. Instead, what is needed is a solution that would allow immediate recognition and handling of problems from a multitude of various modules. Likewise, the design calls for an implementation that conceptually lets each module proceed in parallel (or, at least apparent parallel).

Certainly, the notions of tasks are now motivated. Rather than blindly proposing a problem to students and allowing them to solve it using tasks, they have instead been proposed a problem that can be solved sequentially, but which may have a more correct solution that is non-sequential. Perhaps a design and implementation using tasks could appear as:



Design #3

Even though students are implementing on a sequential machine, the "apparent parallelism" of the design lends itself to using tasking even in a sequential environment. The "inherent complexity" of the problem motivates the use of tasks, even though the programmers should always be consciously aware the solution is probably running on a single CPU rather than actually running upon multiple processors. Unfortunately, even though tasks may be used to design a simple and elegant solution, tasking can cause problems, even when used in a sequential environment.

2.2 Problems in Apparent Parallelism

There are serious problems that befall the first-time task user. First of all, any task that does I/O interfacing with the user (so-called real-time user I/O) can temporarily lock up the entire system on many implementations. This occurs because the task that has issued a **get** or **get_line** is running, and cannot be swapped out until the completion of the I/O. Secondly, a non-preemptive or non-time-slicing run-time environment could cause any one of the above tasks to "hog" the system, effectively locking it up (or worse yet, allowing a less-important task to execute for an indefinite amount of time). Solution? Students need to understand how to implement a pseudo time-slicing algorithm, even on non-time-slicing machines. This can be as simple as the explaining the judicious use of a delay 0.0 to cause the scheduler to look at other *ready* tasks.

The problem is that instructors often start by discussing both a true parallel implementation, and the minimum delays that a program unit is

willing to wait for either a **timed accept** or a **timed entry call.** Instead, students need to discuss and understand both the problems with using time in tasking, and the problems with using multiple processors.

3. REAL-TIME SYSTEMS

When educators start talking about using either timed accepts or timed entry calls, they are now discussing the concepts of placing "deadlines" in processing. This is certainly a good thing, as most real-world applications require deadlines. However, as Ada educators, we often miss most of the important concepts that are required to effectively educate students who can understand the possibilities and limitations of Ada tasking. Do we even understand ourselves how deadline-driven real-time systems are constructed?

3.1 Simple Tasking and Basic Tasking Problems

Simple, no-deadline tasking is the area that most students begin and end with in terms of a "tasking education". In no-deadline tasking, the clock is ignored. Because there are no deadlines of any type, it doesn't matter whether or not you are implementing tasking as concurrent or parallel (uni-processor or multi-processor).

3.1.1 Periodic Tasks

Strictly speaking, a "periodic" or re-occurring task is not a "no-deadline", although it is often treated as such. Even when teaching about "periodic" tasks, there are problems associated with the use of, for example, the **delay 5.0** to cause a task to execute every 5 seconds. First of all, as I am sure we all know, the execution of the task itself takes up some time. This time (let's call it **x**) must be accounted for. Therefore, the delay statement should be **delay 5.0 - x**. However, we must know that **x** is not a constant. Therefore, every time we go through the periodic task, we must calculate a new **x**. An important concept, also, is that if the periodic delay is small, then **x** might eventually get larger than the delay. Since we are talking no-deadline (ignore the clock), this isn't really a problem, just something that students should understand the implications of. If it happens, the overhead of calculating the new **x** each time only further delays the response time, as we are already backed up more than a full cycle.

Another problem is the problem with the resolution of the system clock vs. the accuracy of the delay statement. [Clapp 1990] points out the problems associated with clocks and delays.

Finally, even in no-deadline tasking, problems such as deadlocks and priority inversion crop up.

3.1.2 Deadlocks

Even without a clock-driven deadline to worry about, serious problems can occur in even simple Ada tasking systems. First of all (and most deadly), both deadlocks and livelocks can occur [Coffman 1973]. The problem with teaching Ada tasking too early in the computer science curriculum is that students without an adequate knowledge of operating systems can't fully understand how to prevent, avoid, or detect deadlock.

Granted, a fairly simple explanation of what a deadlock is can be given early on in even a beginning computer science course. However, the actual definition of deadlocks, and, most importantly, how to design systems that avoid deadlocks requires an understand of how a system allocates and assigns resources. This should not be put off until after tasking is taught. Especially in a non-academic environment (i.e. a training environment), a serious discussion of the many different kinds of deadlock should be covered. Often, we brush off this topic with a fairly simple "don't even call a task that could potentially call you before the first rendezvous is complete". Totally insufficient!

Preventing deadlocks is strictly a design function. Even something so simple as declaring a task in the wrong place can "freeze" a system (due to the task/master dependencies Ada must check before allowing program unit to complete). Parallel (or concurrent routine) design is not something given to naive programmers. As educators, we at least need to explain to beginners that great care and experience is needed even when implementing trivial tasking problems. Tasking makes possible a whole new class of ways for beginners to screw up unintentionally. Remember, we don't want to frustrate our students, making them hate Ada before they can even become familiar with it.

3.1.3 Priorities and Priority Inversion

Using priorities in tasking is another subject that, on the surface, appears simple. I tended early on to "brush over" this subject, thinking that nothing could be simpler. The simple fact of the matter is "Ada and priority-driven tasks don't work!". [Locke 1988]. First of all, the FIFO queues that Ada uses to implement entry points forces low-priority tasks ahead of you in line to rendezvous first. Thus, a higher-priority task cannot usurp a lower-priority task ahead of it. This alone is incompatible with true real-time programming. As a matter of fact, people who actually write and maintain real-time systems frequently laugh

immediately when the mechanics of Ada FIFO queues are explained to them.

Along with FIFO queue problems, *priority inversion* occurs when it is impossible to run a high-priority task without running a lower-priority task first. Cornhill and Sha [1987] explain in terms of client tasks and servers. Basically, this occurs in Ada because the *priority inheritance* scheme only occurs between the running server and the client that is being served immediately (the top entry in the queue, the caller that is in rendezvous). Partial solutions can range from implementing intermediate tasks to using a large number of entry points .

Another problem with priorities in Ada is that Ada does not support *dynamic priorities*. It is impossible to change the priority of a task, since the task priority is static, and hard-coded at compile. This, coupled with the FIFO queue that Ada uses, makes serious priority-driven tasking impossible.

To make real-time tasking work at all, some mechanism for avoiding this FIFO dependency must be implemented. This can take the form of "intermediate" tasks that reorder a queue, having multiple "classes" of entry points (one for each priority that could possible call each "class" of entry point within a task), or communicating with the "bare machine" (via assembly language and the predefined package ***machine_code***). Neither of the three is wrong. As Norman Cohen points out [Cohen 1988], tasking sometimes is inherently machine-dependent. Granted, one of the goals of the original Ada design team was to design a portable language. However, one of the goals was also to make a language that was "usable". Tasking, far more than any other feature of the Ada language, can require knowledge not only of the host machine, but of the specific characteristics of the operating system or run-time environment. If this is so, then portability is sacrificed. However, if the initial problem requires a machine-specific solution, then abandoning portability is part of the solution. Indeed, it becomes part of the "inherent complexity".

3.2 Soft-Deadline Tasking (Fault-Tolerance)

Once you get into true real-time programming, you have to start considering the clock in terms of meeting clock-imposed deadlines. Ada is best suited for working with so-called soft deadlines. In soft deadlines, "some" of the imposed deadlines won't be met on time (or at all). Those that are met, but not on time, are allowed to be "occasionally" late by a "reasonable" amount of time. The quotes signify that there is an inherent fuzziness involved. The definition of "reasonable", "some", and "occasionally" depend greatly upon the application, are very subjective, and subject to dynamic constraints. Suffice it to say that careful analysis of the problem are required to fully define the allowable parameters for failure (that is exactly what a missed deadline is, a failure of the system to

meet the needs of the user). Failure, however, can be planned for. In a distributed system, failure can signify that an application (or task) needs to be rehosted or restarted. A serious problem in Ada is that a task cannot easily inform anybody (even its' master) when it dies. Granted, an attempt to call a terminated or completed task will raise an exception. However, it would be nice to know that a task has died BEFORE I attempt to call on it for a rendezvous. Because of this limitation, it is easy to assume that some tasks won't be available when they are called. Proper design of the exception handler, though, could allow a task to be restarted.

A serious problem, though, anytime you start depending upon clock time is the implicit ordering of exactly when an event occurs, and how to recover from errors (exceptions) if a resilient or fault-tolerant environment is needed.

3.2.1 Non-Distributed Environments and Problems

Even when a non-distributed system using concurrent processing is implemented, a real-time system that is trying to meet deadlines can have serious problems. Because deadlines usually imply that some degree of fault-tolerance is required (what good is knowing that you missed a deadline if there isn't some recovery or remedial action available), you have to consider error recovery and fault-masking. Fault masking refers to "hiding" the error or taking remedial action) so that the actual errors disappears. In Ada, any exception handler that does not propagate the exception performs fault masking. However, in cases where a *rollback* is required, Ada does not provide mechanisms for recovery. This is not a deficiency of Ada, as I do not know of any language that does.

Rollbacks refer to the concept of restoring the system to a state that was consistent before an error occurred (and, hopefully, avoiding the same error in the future). This requires state saves (which include checkpointing files, buffering output, taking "snapshots" of memory, and even calling system-dependent routines to determine queue contents) that need to occur at predetermined intervals. The discussion of rollback and recovery is beyond that scope of this paper (see [Lamport 1978], [Koo 1987], and [Hecht 1976]). What is important, however, is that students understand the great complexity that a fault-tolerant deadline-driven system requires, even in a non-distributed environment.

3.2.2 Distributed Environments and Problems

The biggest two problems in a distributed environment are time and location. Ada provides only the minimal mechanisms to determine the time of a particular task [Volz 1987]. Additionally, when we call a

task, there is no way to determine where the task is located. This is very critical, as a detected failure (an exception or non-response) might warrant a restarting via allocating a new value for a task pointer. But, how do we restart on the correct processor? Without the use of system calls (non-portable), this is currently impossible.

Even on a soft-deadline system, exact time is very important. In any distributed system, determining the time is a very demanding task. Effectively, it is impossible to determine a true *global* time. Instead, we can only place a lower bound on it. Often, because of clock accuracy problems, we can't even accurately determine the local time. This is particularly important when we have to roll-back to a time prior to time **X** on processor **Y**. It is difficult to determine the relative local time that corresponds to this time **X**. In any case, clock synchronization becomes a major problem on any distributed system.

3.3 Hard-Deadline Tasking

This area, the most rigorous of the three, is the one that is often least covered and least understood. It is difficult to effectively teach this area in terms of tasking, because of the large amount of "outside" knowledge necessary. In a hard-deadline environment, fault-tolerance and reliability require that 100% of all deadlines termed "critical" must be met (or as close to 100% as possible). Because of these rigid time constraints, system snapshots and saves must be done more frequently the further away from the deadline that you are [Krishna 1984]. As you get closer to the end of the task (and closer to meeting the deadline), there is less work left to do, so less frequent saves are required. This is because the "penalty" of a failure (having to rollback to the last save) decreases depending upon the amount of work left to do. Because of the huge amount of overhead, in terms of scheduling, task communication, and I/O requests for saves, programming languages by themselves typically do not have the power to implement hard-deadline parallel or concurrent processing without resorting to many system calls (and a corresponding lack of portability.)

Without a doubt, Ada does not have the power to support hard-deadline tasking. Experiments on Ada as it currently is show that only 84% of deadlines can be met. Even with several suggested changes to the language, only 89% to 95% of deadlines can be met [Locke 1988]. Clearly, this is insufficient in a critical real-time environment where an extremely high degree of fault-tolerance is required. If Ada is to be used as the language for these types of applications, portable Ada is insufficient. To make the system work, a choice or either system calls or 3rd party add-on boards and hardware is required.

Is this bad? Is resorting to myriad system calls defeating the purpose of Ada? Not at all! Ada was designed to be a general-purpose

real-time language, true. But, any language that tries to be "all things to all users" is probably doomed to failure (look at Pl/1, for example). Instead of indiscriminately using system-dependent calls throughout a system, these calls can be placed logically (and physically) in a rather small kernel or implementation-defined pragmas or subroutines [Cohen 1987]. As Norman Cohen points out, the Ada Run-Time Environment Working Group (ARTEWG) has already defined a provisional set of such interfaces. Although it is unlikely that these interfaces would become common (thus allowing true portability), at least the non-portable code will be well-defined and identifiable.

4. MYTHS ABOUT TASKING

Tasking, in general, is seen by some students as a way to speed up a potentially slow process. I cannot stress enough that, as educators and trainers, we are doing our students a serious injustice if we allow them to feel, in any way, that this is true. Placing parallelism (or concurrence) in a sequential process gains nothing, and actually slows the system down (if not because of any of the potential problems discussed above, then due to the overhead of having to schedule calls and rendezvous). Students often come away from introductory classes in Ada feeling that task calls and accepts are "automatically" handled by the system. Not true! **Tasking is expensive computationally!**

First of all, in most environments, defining stack space for each task eats up precious system resources, and requires a lot of overhead. Secondly, having to implement a scheduling system takes up valuable machine cycles. The CPU must determine who is ready vs. who is blocked, determine who has first-shot at the CPU, and then either make a context-switch or load the next task into memory to execute. These things can eat up time if they are done repeatedly throughout an entire run. The bottom line is this: Tasking should be used only if the design requires it, and only after careful analysis of both the problem and potential (parallel) solution.

If the above were not enough, I would also like to briefly mention several anomalies of parallel processing [Coffman 1973]. These anomalies are very similar to Belady's page-fault anomaly that we learned in basic operating systems. When using deterministic scheduling, it is possible that decreasing the number of tasks can actually increase the overall system execution time. Parallel processing anomalies, on the other hand, show that increasing the number of processors can actually slow down a system. Although counter-intuitive, it is very true. Also, decreasing the processing time of even a single task or relaxing a partial ordering (removing restrictions as to which tasks must run prior to other tasks) may also increase overall run time.

In a system where there is a high degree of interdependence between tasks (many inter-task rendezvous), the communications overhead (especially on a distributed system that has a poorly-implemented connection system) may make additional processors more expensive than any potential benefit justifies. Clearly, careless use of tasks when they are not required not only fails to speed up the eventual solution, but can seriously slow it down.

5. CONCLUSIONS

Does the above paper paint a rather grim and bleak picture of the future of Ada tasking? Not really. First of all, many of the serious shortcomings of the language itself are already being addressed by Ada 9X [1989] and [1990]. Secondly, all this paper has really done is lightly touch on some of the important design issues that must be addressed when writing real-time or fault-tolerant systems, regardless of the language or eventual host machine. This problem is not Ada specific. It's just that we have never had a high-level language this popular before that forced us to consider such potential problems as a normal part of the language!

When teaching good software engineering, we stress to our students that Ada is just a tool, and that it cannot compensate for poor analysis and design. However, when we take a simplistic approach to teaching tasking, we are crippling our students by putting programming "blinders" on them. We give them very high-level (and incomplete) tools, yet we don't equip them with the knowledge to understand the high-level problems, let alone the low-level ones. Teaching tasking should be viewed as part of teaching real-time systems, distributed, fault-tolerant and resilient programming. As a bare minimum, we need even our undergraduate students to see how Ada's tasking paradigms fit into the overall scheme of software engineering. As a minimum, students should leave even an introductory course with the knowledge that tasking does not either identify or solve problems in real-time distributed programming. If the solution requires parallelism, then tasking gives them the necessary power to solve the problem. However, if parallelism is not appropriate, then using tasks "just because they're there" introduce needless complexity and problems.

Teaching tasking as a single part of a language defeats the purpose of having parallel processing as an inherent part of Ada. Instead, good software engineering education and proper training in problem analysis and design gives students the tools to use tasking effectively, and see where tasking contributes (rather than detracts) to an elegant, effective, implementable and correctly-running solution.

As Sha and Goodenough [1990] point out, "... tasking operations, like rendezvous [are] expensive. If the Ada tasking model doesn't work,

use an alternate form of concurrency, based on Ada procedures. Don't expect the compiler to perform any magic". The magic they are talking about is proper design. The compiler won't do it for you, and neither will indiscriminate use of tasks in a non-parallel problem. As Ada educators, we owe it to our students to make sure that we tell them the truth about Ada tasking.

ACKNOWLEDGEMENTS

I would like to thank Professor Udo W. Pooch, Texas A&M University, for his help in proofing and critiquing this article.

REFERENCES

ADA9X 1989 *Ada 9X Project Report*, Office of the Under Secretary of Defense for Acquisition, Wash., D.C.

ADA9X 1990 *Ada 9X Project Report, Supplement 1*, Office of the Under Secretary of Defense for Acquisition, Wash., D.C.

BROOKS, F.P. Jr. 1987 "No Silver Bullet", *Computer*, Vol 20, No 4.

CLAPP, R.M. and MUDGE, T. 1990 "The Time Problem", *SIGAda Ada Performances issues*, Vol 10, No 3.

COFFMAN, E.G. and DENNING, P.J. 1973 *Operating Systems Theory*, Prentice-Hall Inc., Englewood Clifts, N.J.

COHEN, N.H. 1988 "Dependence on Ada Task Scheduling is not Erroneous", *Ada Letters*, Vol 8, No 2.

CORNHILL, D. and SHA, L. 1987 "Priority Inversion in Ada, or What Should be the priority of an Ada Server Task?", *Ada Letters*, Vol 7, No 7.

HECHT, H. 1976 "Fault Tolerant Software for Real-Time Applications", *ACM Computer Surveys*, Vol 8, No 4.

KOO, R. and TORY, S. 1987 "Checkpointing and Rollback Recovery for Distributed Systems", *IEEE Transactions on software Engineering*, Vol SE-13, No 1.

KRISHNA, C.M., SHIN, K.G., and LEE, Y. 1984 "Optimization Criteria for Checkpoint Placement", *Communications of the ACM*, Vol 27, No 10.

LOCKE, D. and SHA, L. and RAJKUMAN, R. and LEHOCZKY, J. 1988 "Priority Inversion and Its' Control: An Experimental Investigation", *SIGAda International Workshop on Real-Time Ada issues*, Vol 8, No 7.

LAMPORT, L. 1978 "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol 21, No 7.

SHA, L. and GOODENOUGH, J.B. 1990 "Real-Time Scheduling Theory and Ada", *IEEE Computer*, Vol 23, No 4.

VOLZ, R.A. and MUDGE, T.N. 1987 "Timing Issues in the Distributed Execution of Ada programs", *IEEE Transactions on Computers*, Vol C-36, No 4.

Awaiting Ada

by

J. Barry Kolb

Prepared for
Fifth Annual Ada Software
Engineering And Training (ASEET) Symposium

Computer Science Department
Ocean County College
Toms River, New Jersey 08753
(908) 255-0357

Abstract

This paper addresses the need to consider software engineering in the early undergraduate curriculum. The perspective is that of a small two year institution working without Ada. The perceived need for change and limitations of the computer science department are described. This is followed by a description of steps taken to prepare students for a future in software engineering.

I. Introduction

This paper relates the steps taken by the computer science department of Ocean County College to prepare students for work and/or additional education in software engineering despite the lack of Ada language facilities. While not suggesting that the Ada deficiency was an advantage to be emulated, it is suggested that by focusing on the real needs of the students progress was made despite the failure to adopt Ada.

Ocean County College (OCC) is a public two year institution located on the central New Jersey shore. The enrollment of about 7,000 full and part-time students is divided between a younger (18 - 24 year old) daytime population and the older (30+) evening population. Of the approximate 225 computer science majors the majority intend to transfer to four year institutions after graduation. Most students work, at least part-time, and the majority (60+%) have traditionally been female. Data for the spring 1990 semester indicates a dramatic drop in the number of female students, this may be an anomaly.

The students verbal and analytical skills are varied, but generally poorly developed. Many students have exceptional difficulty adjusting to the concept of developing a solution rather than retrieving, from the text for example, a solution. At best many students have been taught "boiler plate" techniques in their high school math and computer classes. Students also find that they must take from one to four semesters of math before starting on the department's required courses. This is often true even for those with a good record in high school math.

The computer science curriculum follows the first two years of the ACM model. Transferring students have few problems achieving third year status at four year institutions throughout the country.

II. The Campaign for Ada

Interest in Ada at OCC began in 1986 following the author's encounters with the NASA Space Station Freedom Project and members of ACM's SigAda. While an Ada compiler was not available for the computer science department's systems, efforts were started to plan for an early introduction of Ada. Unfortunately, this planning did not anticipate the strength of opposition to Ada. Progress on the specific issue of incorporating Ada in the curriculum was blocked by this strong opposition. Given an active opposition it proved impossible to reach a consensus on Ada. This, in turn, blocked funding for Ada.

The failure to adopt Ada may be a battle lost, but war won. The support received from groups like SigAda and ASEET demonstrated repeatedly that the efforts at OCC not were taking place in isolation. Many of the improvements at OCC can be traced to contacts with these groups.

III. The Underlying Need

As it became evident that Ada was not going to be introduced in the near future an examination of the curriculum was begun to determine what steps could be taken without Ada to better prepare students. A primary factor in starting this effort was the author's encounters with individuals involved in software development. A recurring theme in these conversations was the difficulty in converting programers to Ada practitioners. It was not the language that was the problem, programming habits were blocking effective software engineering. All too often the origin of these habits could be traced to undergraduate education. Software engineering is not simply writing larger programs, but writing them under real world constraints of time and budget while meeting safety and reliability requirements.[1] A solid foundation was not being laid for the future development of software professionals.

The question seemed obvious; why not teach it right the first time? There were a number of reasons for the current practices. Since software engineering was not considered appropriate in the lower level undergraduate courses, it had been simply ignored. Many techniques and concepts required in software engineering were considered superfluous when programming in the small. Incorporating these concepts would complicate the learning process, at least in the short term. Yet, the longer term, students would find that their undergraduate education had left them just as poorly prepared of software engineering as high school had for college. In addition, faculty experience, textbooks, and the languages available did not support many important software engineering concepts. It became clear that accepting these reasons lead to a curriculum that produced poorly prepared graduates. With or without Ada change was required.

Thus, the underlying philosophy became one of keeping the goal of software engineering always in sight, even if the students were not fully aware of the goal. This is not an attempt to teach software engineering to a first semester student, rather, it is a recognition of the students long term destination. It is, therefore, our responsibility not to erect roadblocks to that ultimate goal.

III. Actions

If software engineering is "the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software,"[2] then early undergraduate education must lay the foundation. Changes had to be made in our courses to build this foundation while providing instruction in the basics of computer science. This had to be accomplished using ISO standard Pascal, not Ada. Standard Pascal had been selected based on a belief in portability and standards. Turbo Pascal, while far more popular, was avoided due to its disregard for the standard.

Experience has shown that far more can be accomplished in support of software engineering than was first expected. Focusing on course content and teaching methods rather than a new language has been beneficial. Students discover for themselves the need for language facilities that Pascal lacks. As

they push our Pascal compiler to its limits, they begin to see how a language might make their task easier, often identifying features found in Ada for themselves.

Changes in our approach can be traced to specific concepts in software engineering. Software engineering is not considered in full, rather we are seeking to prepare the student for the time when they will face software engineering. Our primary goal is to teach solid computer science concepts without building artificial barriers to the future conquest of software engineering. A brief explanation of some of the steps taken follows.

Modularity

Top down design has long been taught in computer science classes. However, traditionally the resulting modules are not discrete program units. The entire program is created in one file sometimes producing a long list of compile time errors, global variables and many other drawbacks. More important, the program is seen by the student as a single unit producing one answer, not a carefully constructed interdependent system. In our first semester course all procedures and functions are written as "external" units. This permits editing and compiling smaller files which the student sees as discrete modules within a larger program. This also permits the student to focus on a single procedure, following the design, code, and test cycle to completion before moving on to another procedure. The key consequence is a change in how the student views software.

Localization

Creating procedures in external files makes passing correct parameters mandatory. In addition local variables cannot be changed inadvertently by calling procedures. Global variables simply do not work. Initially this is more difficult for the students, especially those who have had Pascal in high school. To clarify the purpose of parameters I have borrowed from Ada to require that all formal parameters carry a comment indicating IN, OUT, or INOUT as the parameter's intended purpose. The required comment clearly distinguishes the logical role of parameters overcoming, to some degree, Pascal's weakness in this area. This serves as a reminder to the student of a parameter's intended role. The result is better programs and a better understanding of how to produce the superior program. Careful selection of programming assignments also makes it possible for students to reuse procedures in later assignments. It usually takes only a single example of this to convince students of the superiority of separate procedures.

Abstraction

Keeping the details of a procedure from the programmer is difficult on a small student assignment where a single student writes the entire program. The concept of abstraction is introduced as students learn that they can code and compile a separate procedure then turn away and focus attention on another part of the assignment. By the second semester this can be taken a step further as assignments may call for students to use an instructor

provided set of procedures given only the parameters and any data types required. They are not given access to source code for the procedures. This too is a trying experience for the students, however, once successful understanding follows.

Information Hiding

This follows closely from abstraction, but cannot be implemented directly with our Pascal compiler. It can be simulated by the elementary tactic of reserving for the instructor the right to change the underlying details of the supplied procedures as long as the parameters and outcomes do not change. Students seem to have little problem with the concept of improving procedures without requiring changes in the calling program.

Standards

The importance of standards and style are stressed from the beginning. Students are given a simple coding standard with their first assignment. Even at this early stage most students can appreciate that a uniform standard contributes to program readability, if only for the instructor grading the programs. Invariably some students, usually with a background in high school Pascal, will challenge portions of the coding standards with reasonable alternative approaches. This has been found to be an ideal opportunity to point out that the utility in many aspects of coding standards lies in establishing a uniform approach across a programming team. There may if fact be many "correct" approaches, but one must be selected. A proper place is then established in the students' minds for coding standards.

Discussion of standards on a higher level is woven throughout our courses. The choice of standard Pascal is an attempt to establish our concern early in the learning process. As students move their programs between the department's system and their home computers they quickly understand the value of a standard. Having confronted difficulties with small student assignments the student develops an early appreciation of standards and portability.

IV. Conclusions

The primary evidence of success comes from conversations with graduates who have encountered software engineering courses or work requirements. It is not uncommon to hear our graduates contrasting the ease with which they adapted to Ada and software engineering with their associates difficulties. We have not taught them software engineering; we have tried to provide the foundation of good programming that software engineering education requires.

1. Gibbs, Norman E. The SEI Education Program: The Challenge of Teaching Future Software Engineers. In *Communications of the ACM*. 32, 5 (May 1989), 594 - 605.

2. Humphrey, Watts S. The Software Engineering Process: Definition and Scope. In *Proceedings of the 4th International Software Process Workshop*. (Moretonhampstead, Devon, UK, May 11-13, 1988). New York, NY, ACM SigSoft, 1989.

This Page Left Blank Intentionally

Panel Session

The Future of Ada

Douglas Samuels, Chair

The purpose of this panel is to provide Ada 9X information, status, and future direction to the education and training community. The panelists will provide a brief overview of the 9X process and discuss Ada 9X limitations, constructs not included, special studies, Ada 9X requirements, and Ada bindings to associated standards and the impact of these standards on the Ada 9X process.

Participants in the panel include:

Major Tom Croak	HQ USAF/SC, selected for his knowledge on Ada bindings
Dr. Robert Dewar	NYU, Distinguished Reviewer and master of why things were not included
Dr. Norm Cohen	IBM, Distinguished Reviewer and expert on limitations
Mr. Bill Carlson	Intermetrics, head of the Mapping/Revision Team

This Page Left Blank Intentionally